Comparing the execution of Floyd-Warshell Algorithm on sequential and parallel execution

[1]Amey Jain, [2]Anshul Jain, [3]Arpit Khandelwal, [4] Dhruv Bharadwaj

[1]1MS18CS016, [2]1MS18CS020, [3]1MS18CS028, [4]1MS18CS041

Abstract

Parallel programming represents the next turning point in how software engineers write software. Today, low-cost multi-core processors are widely available for both desktop computers and laptops. As a result, applications will increasingly need to be paralleled to fully exploit the multi-core-processor throughput gains that are becoming available. Unfortunately, writing parallel code is more complex than writing serial code. This is where the OpenMP programming model enters the parallel computing picture. OpenMP helps developers create multi-threaded applications more easily while retaining the look and feel of serial programming. The term algorithm performance is a systematic and quantitative approach for constructing software systems to meet the performance objectives such as response time, throughput, scalability and resource utilization. The performances (speedup) of parallel algorithms on multi-core system have been presented in this paper. The experimental results on a multi-core processor show that the proposed parallel algorithms achieves good performance compared to the sequential.

Introduction

Parallel computers can be roughly classified as Multi-Core and Multiprocessor. A core is the part of the processor which performs reading and executing of the instruction. However as the name implies, Multicore processors are composed of more than one core. A very common example would be a dual core processor. The advantage of a multicore processor over a single core one is that the multi-core processor can either use both its cores to accomplish a single task or it can span threads which divided tasks between both its cores, so that it takes twice the amount of time it would take to execute the task than it would on a single core processor. Multicore processors can also execute multiple tasks at a single time . Performance is the activity of collecting the information about the execution characteristics of a program. One of the parameter to measure performance is the execution time . Hence change in the design from sequential to parallel approach may result in lesser execution time and is demonstrated through coding practices in OpenMP on the case studies: Floyd Warshell algorithm.

Programming In Openmp

OpenMP is an API (application program interface) used to explicitly direct multi-threaded, shared memory parallelism. OpenMP was introduced in 1997 to standardize programming extensions for shared memory machines as shown in figure 1 . In OpenMP the user specifies the regions in the code that are parallel [5]. The user also specifies necessary synchronization like locks, barriers etc. to ensure correct execution of the parallel region. At run time threads are forked for the parallel region and are typically executed in different processors sharing the same memory and address space.

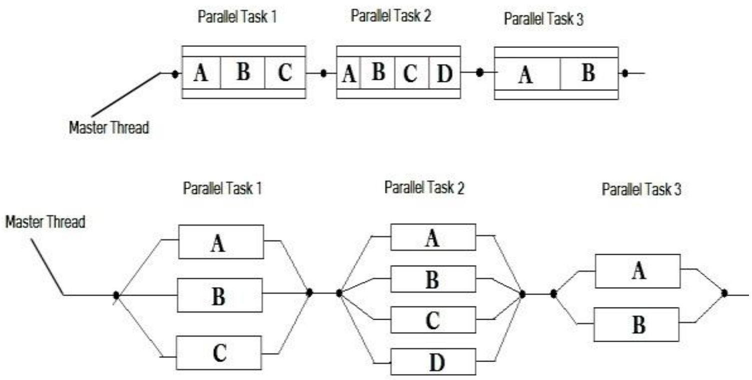


Figure 1: Fork-Join model for OpenMP Program

Advantage of having multiple cores is that we could use these cores to extract thread level parallelism in a program and hence increase the performance of the single program. A lot of research has been done on this area. Many techniques rely on hardware based mechanisms and some depend on compiler to extract the threads .Some the advantages of OpenMP includes: good performance, portable (it is supported by a large number of compilers), requires very little programming effort and allows the program to be paralleled incrementally. OpenMP is widely available and used, mature, lightweight, and ideally suited for multi-core architectures. Data can be shared or private in the OpenMP memory model. When data is private it is visible to one thread only, when data is public it is global and visible to all threads. OpenMP divides tasks into threads; a thread is the smallest unit of a processing that can be scheduled by an operating system. The master thread assigns tasks unto worker threads. Afterwards, they execute the task in parallel using the multiple cores of a processor .

Implementation Details

Objective:

We are implementing the sequential algorithm and parallel algorithm, for this threading concept is used. Program divided into number of threads and each thread is executed independent of other Thread. As number of threads executed simultaneously, time required to execute that program reduces. Main objective of this approach is to save the time required to execute the programs.

Overview of Proposed Work :

We describe the techniques and algorithm involved in achieving good performance by reducing execution time through OpenMP Parallelism on multi-core. We tested the algorithms by writing the program using OpenMP on multi-core system and measure their performances with their execution times as shown in figure 2.

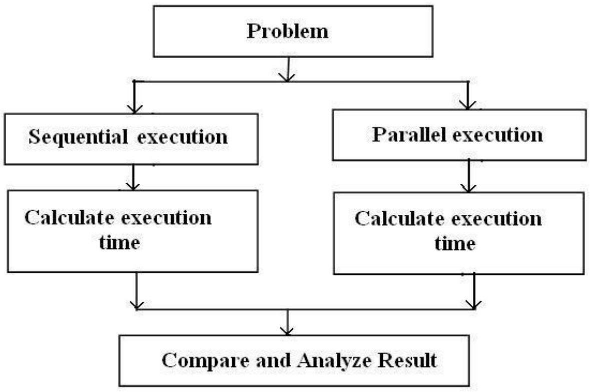


Figure 2: Overview of Proposed Work

Working Modules :

There are some numerical problems which are large and complex. The paper solutions of which takes more time using sequential algorithm on a single processor machine or on multiprocessor machine. The fast solution of these problems can be obtained using parallel algorithms and multi-core system. In this, we select two numerical problems as follows:

Floyd-Warshell Algorithm

Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph. This algorithm works for both the directed and undirected weighted graphs. But, it does not work for the graphs with negative cycles. The Floyd–Warshall algorithm compares all possible paths through the graph between each pair of vertices. It does so by incrementally improving an estimate on the shortest path between two vertices, until the estimate is optimal. In this algorithm each thread has given a chunk size which specifies number of iterations that thread executes. If w(i, j) is the weight of the edge between vertices *i* and *j*, we can define shortestPath(*i*, *j*, *k* + 1) in terms of the following recursive formula:

shortestPath(i, j, 0) = w(i, j);

Algorithm: Floyd_Warshell(int nthreads, int nodes)

Step 1: Start Enter number of thread and number of nodes(n).

Step 2: Initialize matrix
if(i==j) then
mat[i][j]=0 else
generate random between 0-10.

Step 3: Start clock using
start = clock();

Step 4: Set chunk size.

Step 5: Compute shortest path between two vertices in parallel section
#pragma omp parallel for private(i,j)
shared(k) for (i = 0; i < n; ++i){ for (j = 0;
j < n; ++j){
if ((dist[i][k] * dist[k][j] != 0) && (i != j))
if ((dist[i][k] + dist[k][j] < dist[i][j]) || (dist[i][j]==0)){
dist[i][j] = dist[i][k] + dist[k][j];}
}
}

Step 6: End time.

Step 7: Print the time required for computation.

Results

There are two algorithms and each has two versions: sequential and parallel. Both the programs are executed on intel@i5 proessor machine. We analyzed the result and derived the conclusion.

In both the experiment execution time for sequential and parallel program are recorded to compare the results of sequential vs parallel. Execution time is recorded against different dataset to analyzed the speedup of parallel algorithm against sequential. Table 1 shows the time required for parallel matrix multiplication algorithm and sequential algorithm. Table 2 shows the time required for Floyd Warshell parallel and sequential algorithm.

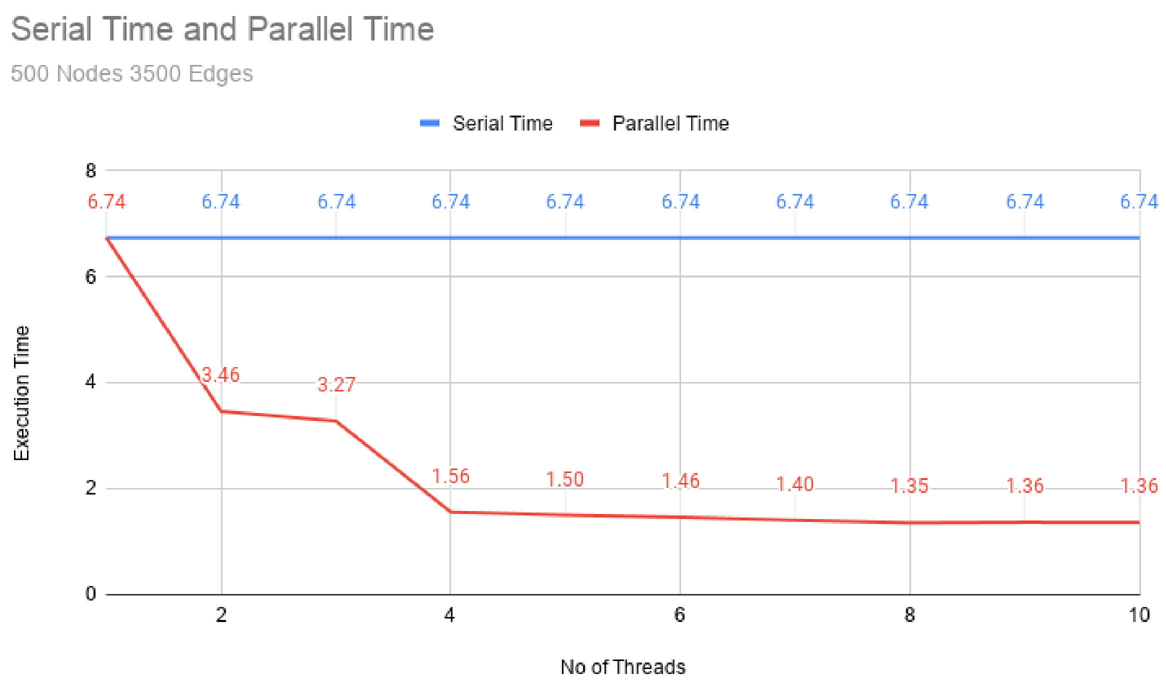
```
for (k = 0; k < V; k++)
{
    for (i = 0; i < V; i++)
    {
        for (j = 0; j < V; j++)
        {
            if (dist[i][k] + dist[k][j] < dist[i][j])
                dist[i][j] = dist[i][k] + dist[k][j];
        }
    }
}
```

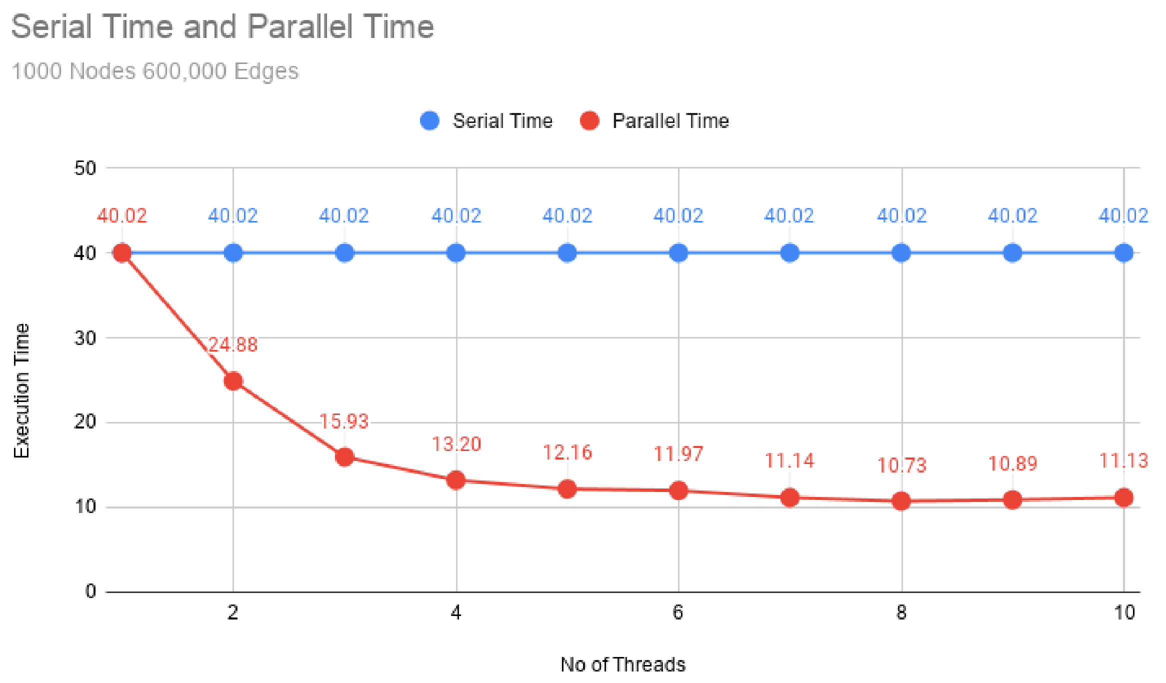
```cpp
#pragma omp parallel shared(dist)
for (k = 0; k < V; k++)
{
	vector<int> *dm = &dist[k];
	#pragma omp parallel for private(i, j) schedule(dynamic)
	for (i = 0; i < V; i++)
	{
		vector<int> *ds = &dist[i];
		for (j = 0; j < V; j++)
		{
			ds->at(j) = min(ds->at(j),ds->at(k)+dm->at(j));
		}
	}
}
```

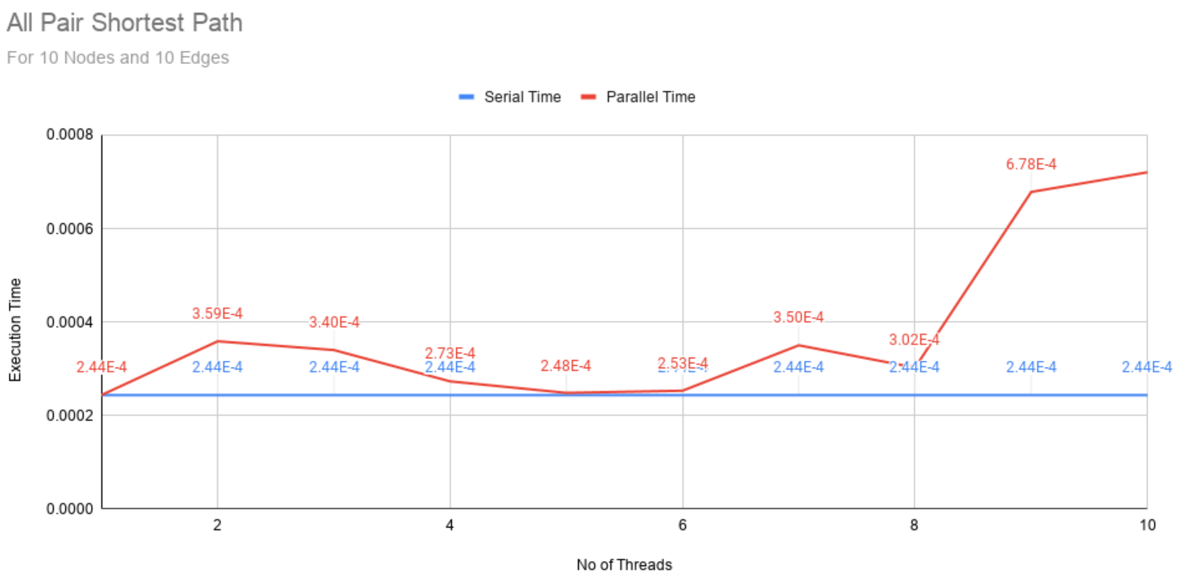
```
amagnum@donut:~/A/5Sem/Labs/PC/Assig2$ g++ -fopenmp apsp.cpp -o apsp
amagnum@donut:~/A/5Sem/Labs/PC/Assig2$ ./apsp < file10_10.in
No. of Nodes: 10
No. of Edges: 10

Total time for Sequential (in sec): 0.00009042

Total time for thread 1 (in sec): 0.00024477
Total time for thread 2 (in sec): 0.00037603
Total time for thread 3 (in sec): 0.00032518
Total time for thread 4 (in sec): 0.00025445
Total time for thread 5 (in sec): 0.00026707
Total time for thread 6 (in sec): 0.00023772
Total time for thread 7 (in sec): 0.00027260
Total time for thread 8 (in sec): 0.00043773
Total time for thread 9 (in sec): 0.00067972
Total time for thread 10 (in sec): 0.00074368

Program Finished...
amagnum@donut:~/A/5Sem/Labs/PC/Assig2$
```

```
amagnum@donut:~/A/5Sem/Labs/PC/Assig2$ ./apsp < file.in
No. of Nodes: 1000
No. of Edges: 600000

Total time for Sequential (in sec): 13.99476180

Total time for thread 1 (in sec): 40.15453781
Total time for thread 2 (in sec): 27.60557904
Total time for thread 3 (in sec): 16.51262690
Total time for thread 4 (in sec): 13.08805827
Total time for thread 5 (in sec): 12.48179030
Total time for thread 6 (in sec): 11.55249141
Total time for thread 7 (in sec): 11.14792959
Total time for thread 8 (in sec): 10.99646994
Total time for thread 9 (in sec): 10.80737062
Total time for thread 10 (in sec): 10.89362506
```

```
amagnum@donut:~/A/5Sem/Labs/PC/Assig2$ ./apsp < file500_3500.in
No. of Nodes: 500
No. of Edges: 3500

Total time for Sequential (in sec): 1.58640130

Total time for thread 1 (in sec): 5.02441602
Total time for thread 2 (in sec): 2.59547270
Total time for thread 3 (in sec): 1.97908531
Total time for thread 4 (in sec): 1.63569454
Total time for thread 5 (in sec): 1.50249481
Total time for thread 6 (in sec): 1.52660898
Total time for thread 7 (in sec): 1.40898608
Total time for thread 8 (in sec): 1.34547545
Total time for thread 9 (in sec): 1.37974935
Total time for thread 10 (in sec): 1.36901064

Program Finished...
amagnum@donut:~/A/5Sem/Labs/PC/Assig2$
```


Serial Time and Parallel Time
500 Nodes 3500 Edges
Serial Time
Parallel Time
Execution Time
No of Threads
6.74
3.46
3.27
1.56
1.50
1.46
1.40
1.35
1.36
1.36


Serial Time and Parallel Time
1000 Nodes 600,000 Edges
Serial Time
Parallel Time
Execution Time
No of Threads
40.02
24.88
15.93
13.20
12.16
11.97
11.14
10.73
10.89
11.13



Observation

1. For smaller size data, multiple threads usually results in increased execution time.
2. For larger size data, multiple threads gives good results, here execution time gets decreased.
3. There is a limit to which execution time can be improved, after a certain number the process management becomes more and hence it affects the execution times of the problem.
4. Somewhere between 4 - 8 threads performs best in this particular problem.

Conclusion

The algorithms with small data set gives good performance when executed by a sequentially programming. But 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.

REFERENCES

[1] Daniel Lorenz, Peter Philippen, Dirk Schmidl and Felix Wolf. "Profiling of OpenMP Tasks with Score-P". *International Conference on Parallel Processing Workshops*, German Research School for Simulation Sciences, 52062 Aachen, Germany, 2012.

[2] D. Dheeraj, B. Nitish, Shruti Ramesh, "Optimization of Automatic Conversion of Serial C to Parallel OpenMP", *International Conference on Cyber-Enabled Distributed Computing and Knowledge Discover*, PES Institute of Technology Bangalore, India, Dec 2012.

[3] Suneeta H. Angadi, G. T. Raju Abhishek B, "Software Performance Analysis with Parallel Programming Approaches", *International Journal of Computer Science and Informatics*, *ISSN (PRINT): 2231 -5292*, *Vol-1*, *Iss-4*, 2012.

[4] Sanjay Kumar Sharma, Dr. Kusum Gupta, "Performance Analysis of Parallel Algorithms on Multi-core System using OpenMP Programming Approaches", *International Journal of Computer Science*, *Engineering and Information Technology (IJCSEIT)*, *Vol.2*, *No.5*, October 2012.

[5] Han Cao1a, Fei Wangb, Xin Fang, Hong-lei Tu, "OpenMP Parallel Optimal Path Algorithm and Its Performance Analysis", *Jun Shi World Congress on Software Engineering*, *DOI 10.1109*, WCSE, 2009.

[6] Javier Diaz, Camelia Munoz-Caro, and Alfonso Nino, "A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era" *IEEE Transactions on Parallel and Distributed Systems*, *Vol. 23*, *No. 8*, AUG 2012.

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

[8] Paul Graham, Edinburgh, "A Parallel Programming Model forShared Memory Architectures", *Parallel Computing Center*, The University of Edinburgh, March 2011.