IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661, p- ISSN: 2278-8727Volume 16, Issue 2, Ver. X (Mar-Apr. 2014), PP 58-62
www.iosrjournals.org

"Performance Analysis of Parallel Algorithm over Sequential using OpenMP"

Pranav Kulkarni[1], Sumit Pathare[2]
[1](IT Department, SITRC Nashik, India)
[2](IT Department, SITRC Nashik, India)

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.*
Keywords: *Multi-core, Multiprocessor, OpenMP, Parallel programming.*

I. 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 [1]. 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 [2]. 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: Matrix Multiplication and Floyd Warshell algorithm. Rest of the paper is organized as follows. Section 2 giving detailed description of related work, section 3 explains programming in openmp, section 4 is about detailed implementation details of algorithms, section 5 focuses on result and analysis, paper is concluded by mentioning future research in the field.

II. Related Work

In 2009, Han Cao1a, Fei Wangb, Xin Fang, Hong-lei Tu [5] proposed the idea to apply the non hierarchical algorithm such as Dijkstra's algorithm to different levels and the entrance points and exit points (node E) between high-level and low-level are obtained by the heuristic directing search approach. The algorithm procedure to find the satisfactory path, in terms of the minimum travel time based on the Manhattan distance and travel speed associated with the edges in the network.

In 2012, Songmin Jia, Xiaolin Yin, and Xiuzhi Li [7] proposed an effective Simultaneous Localization and Map Building (SLAM) technique for indoor mobile robot navigation based on OpenMP. Particle Filter (PF) based SLAM provides an effective indoor mobile robot navigation framework, but real-time performance of PF needs improving due to their inherent complex and intensive computation. OpenMP is the product of the multi-core technology development and has been widely accepted by both industry and academia. We propose a multi-thread particles filter algorithm based on OpenMP to reduce computation time of PF and execution time of SLAM. The results in real experiments and simulations show that the parallel PF-SLAM algorithm based on OpenMP could reduce the SLAM execution time while guaranteeing the accuracy of SLAM.

III. 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 [2]. 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.

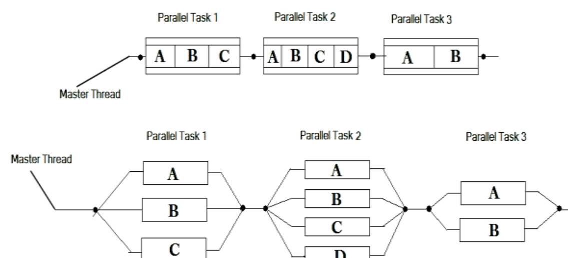

Figure 1: Fork-Join model for OpenMP Program

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 [1].

IV. Implementation Details

4.1 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.

4.2 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.

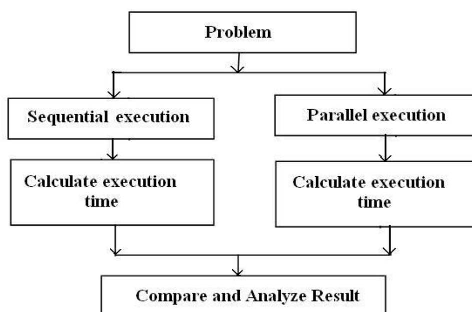

Figure 2: Overview of Proposed Work

4.3 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:

4.3.1 Matrix Multiplication

In matrix multiplication algorithm, there is no task dependency hence thread and kernel instances parallel running reduces execution time. Matrix multiplication problem also solved using openMP. It will give best result by executing it sequentially rather than parallel when number of rows and columns are less, but as number of rows and columns (i.e matrix A[500][500]) increases sequential algorithm's performance decreases where role of parallel programming comes into play. Algorithm for it given as follows:

Matrix_Multiplication(int size, int n)

Step 1: Size represents size of matrix and n represents number of threads.

Step 2: Declare variables to store allocated memory

Step 3: Declare variables to input matrix size and variables to be used by OpenMP functions a_r, a_c, b_r, b_c, nthreads, tid, chunk.

Step 4: Declare variable to calculate the time difference between the parallelism.

Step 5: Accept number of rows and columns.

Step 6: Allocate memory for matrix one

Step 7: Allocate memory for matrix two

Step 8: Allocate memory for sum matrix

```
{
c=(int *) malloc(10*a_r)
for( i=0;i< b_c; i++)
{
c[i]=(int *) malloc(10*b_c)
}
}
```

Step 9: Start the timer

```
double start = omp_get_wtime()
```

Step 10: Here Actual Parallel region starts #pragma omp parallel shared(a,b,c,nthreads,chunk)

```
private(tid,i,j,k)
{
tid = omp_get_thread_num()
if (tid == 0)
{
nthreads = omp_get_num_threads()
print Starting matrix multiple example with number of threads
}
```

Step 11: Initializing first matrix.

Step 12: Initializing second matrix.

Step 13: Print Thread starting matrix multiply.

```
#pragma omp for schedule (static, chunk)
for(i=0;i<a_r; i++)
{
        for(j=0;j<a_c; j++)
        {
        for(k=0;k<b_c; k++)
        {
        c[i][j]=c[i][j]+a[i][k]*db[k][j]
        }
        }
}
```

Step 14: end the timer

```
double end = omp_get_wtime( )
```

Step 15: Store the difference

```
dif = end – start
```

Step 16: Free memory

```
for(i=0;i<a_r; i++)
{       free(a)
        free(b)
        free(c)
}
```

Step 17: Print the time required for computation.

4.3.2 Floyd-Warshall Algorithm

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.
```

V. Experiment Results

There are two algorithms and each has two versions: sequential and parallel. Both the programs are executed on intel@i3 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.

Table 1: Comparison Chart for Matrix Multiplication Algorithm

Data Set	Sequential program	Parallel Program with 2threads	Parallel Program with 4 Thread
200	0.0555	0.0529	0.0435
400	0.4236	0.3212	0.3507
600	1.5991	1.1313	1.2182
800	5.9626	3.3564	3.0578

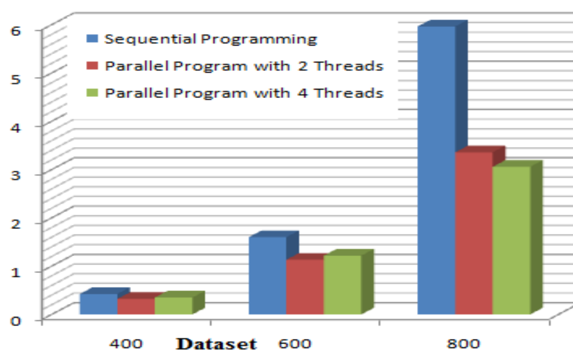

Figure 3: Performance Analysis of Matrix Multiplication Algorithm

Table 2: Comparison Chart for Floyd Warshell Algorithm

Data Set	Sequential program	Parallel Program with 2threads	Parallel Program with 4 Thread
50	0.00286	0.00320	0.00467
60	0.00457	0.00345	0.00772
70	0.00729	0.00518	0.00504
80	0.01079	0.00735	0.00702

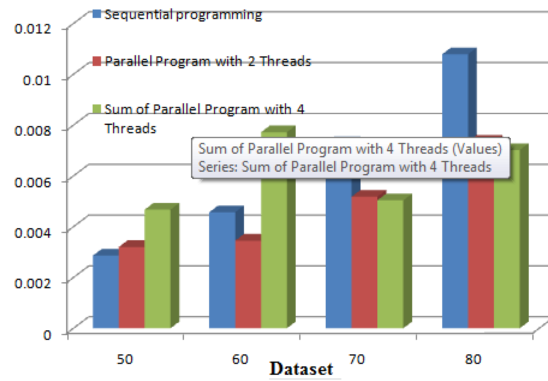

Figure 4: Performance Analysis of Floyd Warshell Algorithm

From Fig. 3 and 4 we can see initially sequential algorithm requires less time than parallel but as dataset increases performance of parallel algorithm increases.

VI. 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.

Acknowledgements

The authors want to thank their guide Prof. Vijay R. Sonawane and Prof. Vivek N. Waghmare for invaluable assistant that we have received from them.

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.