# Performance Analysis of Parallel Algorithms on Multi-core System using OpenMP

Sanjay Kumar Sharma[1], Dr. Kusum Gupta[2]

[1]Departemtne of Computer Science, Banasthali University, Rajasthan, India
skumar2.sharma@gmail.com
[1]Departemtne of Computer Science, Banasthali University, Rajasthan, India
gupta_kusum@yahoo.com

## *Abstract*

*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 π 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.*

## *Keyword*

*Multi-core Architecture, OpenMP, Parallel algorithms, Performance analysis.*

## 1. Introduction

To see the parallelism in the developed applications, a number of tools available for them. Multithreading is the technique which is allow to execute multiple threads in parallel. The computer architecture has been classified into two categories: instruction level and data level. This classification is given by Flynn's taxonomy [1]. Performance of parallel application can be achieved using Multi-core technology.

Multi-core technology means having more than one core inside a single chip. This opens a way to the parallel computation, where multiple parts of a program are executed in parallel at same time [2]. The factor motivated the design of parallel algorithm for multi-core system is the performance. The performance of parallel algorithm is sensitive to number of cores available in the system, core to core latencies, memory hierarchy design, and synchronization costs. The software development tools must abstract these variations so that software performance continues to obtain the benefits of the Moore's law.

In multi-core environment the sequential computing paradigm is not good and inefficient, while the usual parallel computing may be suitable. One of the most important numerical problems is solution of system of linear equations. Systems of linear equations arise in the science domain such as fusion energy, structural engineering and method of moment formulation of Maxwell equation.

## 2. Related work

With the invention of multi-core technology, the parallel algorithm can get the benefits to improve the performance of the application. Multi-core technologies supports multithreading to executing multiple threads in parallel and hence the performance of the applications can be improved. We studied the number of algorithms on multi-core/parallel machines and the performance metrics of numerical algorithms [3], [4].The research on hybrid multi-core and GPU architectures are also emerging trends in the multi-core era [5].

In order to achieve the high performance in the application, we need to develop the correct parallel algorithm, requires the hardware and the language like OpenMP. The OpenMP has the support of multithreading. The program can be developed so that all the processor can be busy to improve the performance. So far all the works discuss about the performance of the algorithms. Our unique approach is to find the performance of some numerical algorithms on Multi-core system using OpenMP programming techniques.

## 3. Overview of Proposed Work

There are some numerical problems which are large and complex; 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 paper we select two numerical problems. The first problem is to approximately compute the value of $\boldsymbol{\pi}$ using method of integration, and second is solution of system of linear equations [6]. We describe the techniques and algorithm involved in achieving good performance by reducing execution time through *OpenMP* Parallelization 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.

In our proposed work, we estimate the execution time taken by the programs of sequential and parallel algorithms and also computed the speedup. The schematic diagram of our proposed work for the solution of system of linear equation is shown in Fig.1.

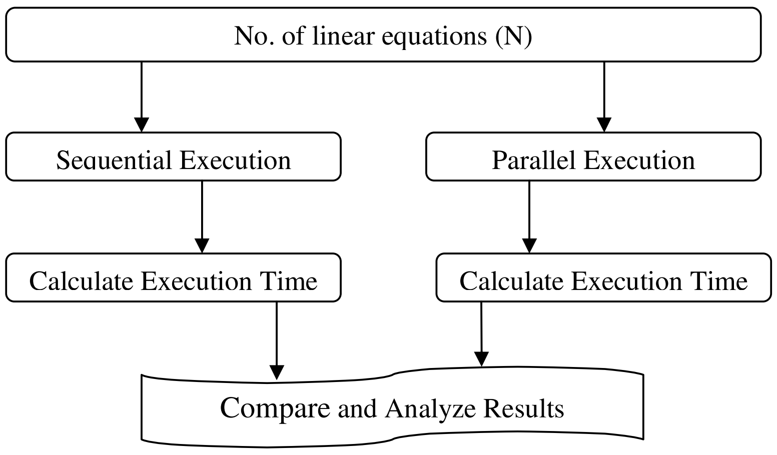

Figure 1 Modules of parallel Algorithm

## 4. Programming in OpenMP

An OpenMP Application Programming Interface (API) was developed to enable shared memory parallel programming. OpenMP API is the set of compiler directives, library routines, and environment variables to specify shared-memory parallelism in FORTRAN and C/C++ programs [7]. It provides three kinds of directives: *parallel work sharing, data environment and synchronization* to exploit the multi-core, multithreaded processors. The OpenMP provides means for the programmer to: create teams of thread for parallel execution, specify how to share work among the member of team, declare both shared and private variables, and synchronize threads and enable them to perform certain operations exclusively [7].

OpenMP is based on the fork-and-join execution model, where a program is initialized as a single thread named master thread [7]. This thread is executed sequentially until the first parallel construct is encountered. This construct defines a parallel section (a block which can be executed by a number of threads in parallel). The master thread creates a team of threads that executes the statements concurrently in parallel contained in the parallel section. There is an implicit synchronization at the end of the parallel region, after which only the master thread continues its execution [7].

### 3.1 Creating an OpenMP Program

OpenMP's *directives* can be used in the program which tells the compiler which instructions to execute in parallel and how to distribute them among the threads [7]. The first step in creating parallel program using OpenMP from a sequential one is to identify the parallelism it contains. This requires finding instructions, sequences of instructions, or even large section of code that can be executed concurrently by different processors. This is the important task when one goes to develop the parallel application.

The second step in creating an OpenMP program is to express, using OpenMP, the parallelism that has been identified [7]. A huge practical benefit of OpenMP is that it can be applied to *incrementally* create a parallel program from an existing sequential code. The developer can insert directives into a portion of the program and leave the rest in its sequential form. Once the resulting program version has been successfully compiled and tested, another portion of the code can be parallelized. The programmer can terminate this process once the desired speedup has been obtained [7].

## 4. Performance of Parallel Algorithm

The amount of performance benefit an application will realize by using OpenMP depends entirely on the extent to which it can be parallelized. ***Amdahl's*** law specifies the maximum speed-up that can be expected by parallelizing portions of a serial program [8]. Essentially, it states that the maximum speed up (*S*) of a program is

$$\mathbf{S = 1/\,(1\text{-}F) + (F\,/\,N)}$$

where, F is the fraction of the total serial execution time taken by the portion of code that can be parallelized and N is the number of processors over which the parallel portion of the code runs.

The metric that have been used to evaluate the performance of the parallel algorithm is the speedup [8]. It is defined as

$$Sp = T_1 / T_P$$

Where, $T_1$ denotes the execution time of the best known sequential algorithm on single processor machine, and *Tp* is the execution time of parallel algorithm on *P* processor machine.

In other word, *speedup* refers to how much the parallel algorithm is faster than the corresponding sequential algorithm. The linear or ideal speedup is obtained when *Sp=P*.

## Proposed Methodology

### 5.1 Calculation of $\pi$

The function $f(x) = \frac{1}{1+x^2}$ can be used to approximate the value of $\pi$ using numerical integration. Consider the evaluation of definite integral

$$I = \int_0^1 \frac{1}{1+x^2} dx = \arctan(1) - \arctan(0) = \frac{\pi}{4}$$

where, *f(x)* is called the integrand, *a* lower, and *b* is upper limits of integration.

Integration of function *f(x)* can be evaluated numerically by splitting interval [a, b] into *n* equally spaced subintervals. We assume that the intervals are constant and its interval width is *h= (b-a)/n*. Method of integration (Simpson 1/3 rule) have been used to approximately compute the values of $\pi$. The formula and its sequential algorithm are given below.

$$\int_a^b f(x)dx \approx \frac{h}{3}\left[ f(x_0) + 4\sum_{\substack{i=1 \\ i=odd}}^{n-1} f(x_i) + 2\sum_{\substack{i=2 \\ i=even}}^{n-2} f(x_i) + f(x_n) \right]$$

**Sequential Algorithm**

```
1.  Input N   // N no. of intervals, a=0 and b=1 are limits
2.  h = (b-a) / N
3.  sum= f(a) + f(b) + 4*f(a+h)
4.  i = 3 to N-1 step +2 do
5.      x = 2*f (a+ (i-1) *h + 4 * f (a+ i*h)
6.      sum = sum+ x
7.  End loop [i]
8.  Int = (h/3) * sum
9.  Print 'Integral is = ', Int
```

### 5.2 Solution of System of Linear equations

Solution of system of linear equations is assignment of value to variables that satisfy the equations. To solve the system of linear equations, we considered the direct method: *Gaussian elimination*. It is a numerical method for solving the system of linear equations *AX = B*, where *A* is a known matrix of size *n*×*n*, *X* is the required solution vector, and *B* is a known vector of size n. Consider the n linear equation in n unknowns as

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n &= a_{1,n+1} \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n &= a_{2,n+1} \\ a_{31}x_1 + a_{32}x_2 + \dots + a_{3n}x_n &= a_{3,n+1} \end{aligned}$$

$$\begin{array}{ccccccc} \dots & \dots & \dots & \dots\ \dots & \dots & \dots & \dots \\ a_{n1}x_1 & + a_{n2}x_2 & + & \dots + & a_{nn}x_n & = a_{n,n+1}, \end{array} \qquad (1)$$

where, $a_{i,j}$ and $a_{i,j+1}$ are known constants and $x_i$'s are unknowns.

The parallel algorithm, which is used to solve dense system of linear equations using Gaussian elimination with partial pivoting, is developed. Gauss method is based on transformation of linear equations, which do not change the solution [9] [10][11]. It includes the transformations: *Multiplication of any equation by a nonzero constant, Permutation of equations, Addition of any system of equation to other equation.* This algorithm consists of two phases:

1. In the first phase the Pivot element is identified as the largest absolute value among the coefficients in the first column. Then Exchange the first row with the row containing that element. Then eliminate the first variable in the second equation using transformation. When the second row becomes the pivot row, search for the coefficients in the second column from the second row to the nth row and locate the largest coefficient. Exchange the second row with the row containing the largest coefficient. Continue this procedure till (n-1) unknowns are eliminated [11].
2. The second phase in known as backward substitution phase which is concerned with the actual solution of the equations and uses the back substitution process on the reduced upper triangular system [11].

The sequential algorithm of gauss elimination is given below:

***Sequential algorithm***

```
1.  Input: Given Matrix a [1: n, 1: n+1]
2.  Output: x [1: n]
3.  // Traingularization process
4.  for k = 1 to n-1
5.  Find pivot row by searching the row with greatest absolute value among the elements of
    column k
6.     swap the pivot row with row k
7.      for i = k+1 to n
8.              $m_{i,k} = a_{i,k} / a_{k,k}$
9.           for j = k to n+1
10.              $a_{i,j} = a_{i,j} - m_{i,k} * a_{k,j}$
11.          End loop [j]
12.      End loop [i]
13. End loop [k]
14. // Back substitution process
15.     $x_n = a_{n,n+1} / a_{n,n}$
16. for i = n-1 to 1 step -1 do
17.      sum = 0
18.   for j = i+1 to n do
19.      $sum = sum + a_{i,j} * x_j$
20.    End loop [j]
21.    $x_i = ( a_{i,n+1} - sum )/a_{i,i}$
22. End loop[i]
```

## 5. Design and Implementation

First we study the typical behavior of sequential algorithms and identified the section of operation that can be executed in parallel. In designed parallel algorithm we have used the #pragma directive which shows that the iteration of loop will execute in parallel on different processors. The parallel algorithms for both the problem is given below.

We computed the value of Pi using numerical integration. There are six level of parallelization of numerical integration. We have used the most efficient integration formula level to compute the value of definite integral. In the parallel algorithm we inserted the #pragma directive to parallelize the loop.

**Parallel Algorithm**- Computation of Pi Value

```
1.  Input N   // N no. of intervals, a=0 and b=1 are limits
2.  h = (b-a) / N
3.  Start the clock
4.  sum= f(a) + f(b) + 4*f(a+h)
5.  Insert #pragma  directive to parallelize the loop i
6.  i = 3 to N-1 step +2 do
7.      x = 2*f (a+ (i-1) *h + 4 * f (a+ i*h)
8.      sum = sum+ x
9.  End loop [i]
10. Int = (h/3) * sum
11. Stop clock
12. Display time taken and vale of Int
```

In the sequential algorithm of Gauss elimination we found that the innermost loops indexed by **i,** and **j** can be executed in parallel without affecting the result. In the parallel algorithm, we insert the *#pragma* directive to parallelize the loops.

**Parallel algorithm-** Gauss elimination

```
1.  Input: a [1: n, 1: n+1]   // Read the matrix data
2.  Output: x [1: n]
3.  Set the numbr of threads
4.  Strat clock
5.  // Traingularization process
6.  for k = 1 to n-1
7.     Insert #pragma  directive to parallelize
8.      for i = k+1 to n
9.          $m_{i,k} = a_{i,k} / a_{k,k}$
10.      for j = k to n+1
11.          $a_{i,j} = a_{i,j} - m_{i,k} * a_{k,j}$
12.       End loop [j]
13.     End loop [i]
14.  End loop [k]
15. // Back substitution process
16.  $x_n = a_{n,n+1} / a_{n,n}$
17.   for  i = n-1 to 1 step -1 do
18.       sum = 0
```

```
19.   for j = i+1 to n do
20.       sum = sum + $a_{i,j}$ * $x_j$
21.   End loop [j]
22.  $x_i$ = ( $a_{i,n+1}$ - sum )/$a_{i,i}$
23. End loop[i]
24. Stop clock
25. Display time taken and solution vector
```

## 6. Experimental Results

There are two version of algorithm: sequential and parallel. The programs are executed on Intel@Core2-Duo processor machine. We analyzed the performance using results and finally derived the conclusions. The Intel C++ compiler 10.0 under Microsoft Visual Studio 8.0 used for compilations and executions. The Intel C++ compiler supports multithreaded parallelism with */Qopenmp* flag. The Origin6.1 software is used to plot the graph using the data obtained by the experiments.

In the first experiment the execution times of both the sequential and parallel algorithms have been recorded to measure the performance (speedup) of parallel algorithm against sequential. We used π value from *Mathematica* to compare the accuracy of computed value. *Mathematica* is known for its capability to do computations with arbitrary precision.

The data presented in Table 1 represents the execution time taken by the sequential and parallel programs, difference between m*athematica's* values of π and the speedup. We plot the graph using the data in Table 1 to analyze the performance of parallel algorithm which is shown in fig. 2. The result obtained shows a vast difference in time required to execute the parallel algorithm and time taken by sequential algorithm. The parallel algorithm is approximately twice faster than the sequential.

Table 1 Performance comparison of sequential and parallel algorithm to compute the value of π

| Sr. No. | No. of Intervals | Sequential Execution Time & Difference between results | | Parallel Execution Time & Difference between results | | Performance/ Speed up (S) |
|---|---|---|---|---|---|---|
| | | Time (m. s.) | Difference | Time (m. s.) | Difference | |
| 1 | 1000 | 0 | $-1.67*10^{-7}$ | 0 | $-1.67*10^{-7}$ | 0 |
| 2 | 10000 | 0 | $-1.67*10^{-9}$ | 0 | $-1.67* 10^{-9}$ | 0 |
| 3 | 100000 | 15 | $-1.67*10^{-11}$ | 8 | $-1.67*10^{-11}$ | 1.875 |
| 4 | 1000000 | 78 | $-4.44*10^{-16}$ | 40 | $-2.04*10^{-16}$ | 1.950 |

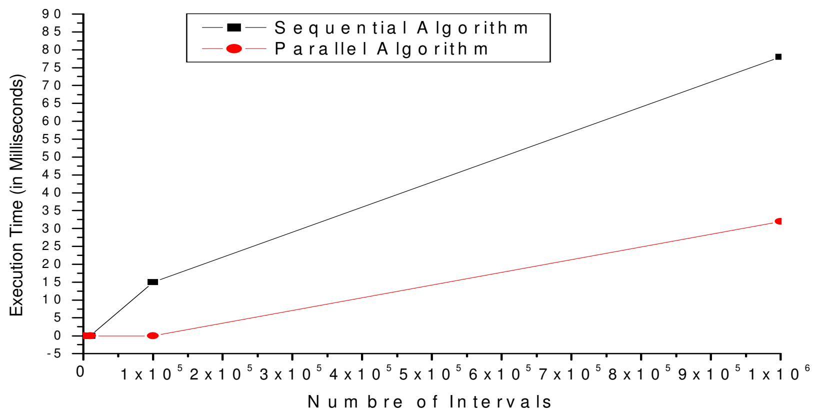

Figure 2 Execution times of sequential and parallel computation of π

In the second experiment, we implemented the sequential and parallel algorithms for finding the solution of system of linear equations. We tested both the algorithms on the equations of different sizes and recorded their execution times. In the program we used the function *timeGetTime()* to calculate the time taken by sequential and parallel algorithms.

The data in Table 2 represent the execution times taken by sequential and parallel programs for the solution of system of linear equations of different sizes, and the speedup. The result shows that the parallel algorithm is efficient than their corresponding sequential algorithm. We plot the graph using data in Table 2 to analyze the performance (speedup) of parallel algorithm which is presented in Fig.3. It shows that the parallel algorithm save significant amounts of execution time and gives more efficient results. The speedup of parallel algorithm on average is approximately twice than their corresponding sequential algorithm**.**

Table 2 Performance comparison of sequential and parallel algorithms

| No. of Equations | Sequential Execution Time (m. s.) | Parallel Execution Time (m. s.) | Performance / Speedup(S) |
|---|---|---|---|
| 100 | 0 | 0 | 0 |
| 125 | 31.5 | 16.50 | 1.90 |
| 150 | 46.5 | 24.25 | 1.91 |
| 175 | 70.5 | 36.50 | 1.93 |
| 200 | 124.5 | 63.50 | 1.96 |
| 225 | 183.25 | 92.25 | 1.98 |
| 250 | 249.5 | 125.25 | 1.990 |
| 275 | 327.5 | 165.50 | 1.97 |
| 300 | 379.25 | 190.00 | 1.996 |

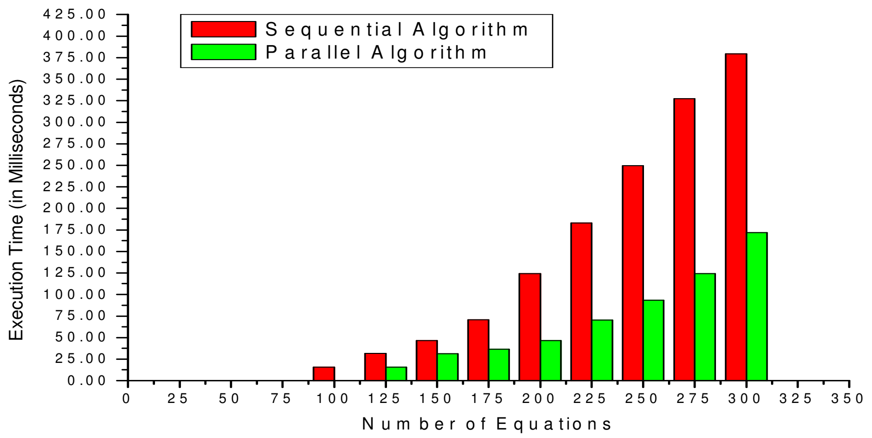

Figure 3. Execution time of sequential and parallel algorithm of Gauss elimination algorithms

## 7. Conclusions and Future Enhancement

In this work we studied how OpenMP programming techniques are beneficial to multi-core system. We also computed the value of Pi and solved linear equations using OpenMP to improve performance by reducing execution time. We also presented the execution time of both serial and parallel algorithm for computation of Pi value and for the solution of system of linear equations. The work has successfully computed the value of Pi and solution of system of linear equation using OpenMP on multi-core machine.

Based on our study we arrive at the following conclusions: (1) we see that parallelizing serial algorithm using OpenMP has increased the performance. (2) For multi-core system OpenMP provides a lot of performance increase and parallelization can be done with careful small changes. (3) The parallel algorithm is approximately twice faster than the sequential and the speedup is linear.

The future enhancement of this work is highly creditable as parallelization using OpenMP is gaining popularity. This work will be carried out in near future for the real time implementation over a large extent and for the high performance systems.

## References

[1] Noronha R., Panda D.K., “Improving Scalability of OpenMP Applications on Muti-core Systems Using Large Page Support, *IEEE Computer, 2007.*

[2] Kulkarni, S. G., “Analysis of Multi-Core System Performance through OpenMP”, *National Conference on Advanced Computing and Communication Technology, IJTES, Vol-1. No.-2, Page. 189-192, July – Sep 2010.*

[3] Gallivan K. A., Plemmons R. J., and Sameh A. H.,"Parallel algorithms for dense linear algebra computations," *SIAM Rev., vol. 32, pp. 54-135, March 1990.*

[4] Gallivan K. A., Jalby W., Malony A. D., and Wijshoff H. A. G., "Performance prediction for parallel numerical algorithms.," *International Journal of High Speed Computing, vol. 3, no. 1, pp. 31-62, 1991.*

[5] Horton M., Tomov S., and Dongarra J., "A class of hybrid LAPACK algorithms for multi-core and GPU architectures," *in Proceedings of the 2011 Symposium on Application Accelerators in High-Performance Computing, SAAHPC '11, (Washington, DC, USA), pp. 150-158, IEEE Computer Society, 2011.*

[6] Wilkinson, B., Allen, M., "*Parallel Programming*", Pearson Education (Singapore), 2002.

[7] Barbara, C., Jost, G., Pas, R.V., "*Using OpenMP: portable shared memory parallel programming*", The MIT Press, Cambridge, Massachusetts, London, 2008.

[8] Quinn, M. J, "*Parallel Programming in C with MPI and OpenMP*", McGraw-Hill Higher Education, 2004.

[9] Angadi, Suneeta H., Raju, G. T. & 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.*

[10] Bertsekas, Dimitri P., Tsitsiklis, John N.,"*Parallel and Distributed Computation: Numerical Methods*", Massanchusetts Institute of Technology, Prentice-Hall, Inc., in 1989

[11] Vijayalakshmi, S., Mohan, R., Mukund, S., Kothari, D.P. ," LINPACK: Power-Performance Analysis of Multi-Core Processors Using OpenMP, *International Journal of Computer Applciation, Vol. 42-No. 1, April 2012.*