International Journal of Advance Research in Science and Engineering
Volume No.07, Special Issue No.04, March 2018
www.ijarse.com
IJARSE
ISSN: 2319-8354

Performance Evaluation of Parallel Algorithm on Multi Core System Using Open MP

Taniya Ashraf Dar [1], Salma Fayaz [2], Afaq Alam khan [3]

Department of Information Technology, Central University of Kashmir, J&K, India[1]

Department of Information Technology, Central University of Kashmir, J&K, India[2]

Asst. Professor, Department of Information Technology, Central University of Kashmir, J&K, India[3]

ABSTRACT

Parallel computing is a form of computation that allows many instructions in a program to run simultaneously, in parallel. This is achieved when a program is to be split up into independent parts so that each processor can execute its part of the program with the other processors, simultaneously. The use of multi core processor is very much dependent on the algorithms used and their implementation. In parallel computation the gains are limited by some overheads that cause parallel codes to run slower. It is beneficial to do a quick calculation with the speed-up equations to establish whether making the codes parallel will actually achieve a significant improvement on the speeds. Parallel programming is not an easy task, using more processors to run the code does not always result in a significant speed-up. In this paper we implement the matrix concatenation algorithm using multithreading (OpenMP). OpenMP is an Application Program Interface (API) that is used to explicitly direct multi-threaded, shared memory parallelism. OpenMP has extensive support for parallelizing loops. Parallel algorithm achieved a significant speed up on multi-core systems, shown from experimental results. This paper evaluates the performance of parallel implementation of matrix concatenation algorithm and compares the same with that of sequential implementation.

Keywords:* *Multi Core Systems, OpenMP, Parallel Computing, Parallelize Matrix Concatenation algorithm, Performance Analysis, Speedup

I. INTRODUCTION

A serial program runs on a single processor, typically the instructions are executed one after the other, in series. In serial programming only one instruction is executed at a time. However, parallel computing is a standard way of utilizing several computer resources for computational scientists and engineers to solve a computational problem [4]. The parallel programs solve a particular problem by divide and conquer technique and obtain the solution to a given problem by dividing it into sub problems, solving these sub problems and combining their solutions to get the solution of the whole problem. The main aim of parallel computing is to reduce the amount of time it takes to run, to solve complex problems and efficient utilization of parallel hardware. Multi-core processors open a way to the parallel computation, by employing more than one core to solve a problem [2]. In Multi-core processors multiple parts of a program are executed in parallel at the same time. In order to improve

processor performance, Thread-level parallelism could be a well-known strategy [6]. There are many platforms available for parallel computing such as OpenCL, MPI, OpenMP, etc, with which we can improve system performance by dividing the task and distributing them among the processors.

This paper is organized as follows. Section II contains the introduction of Parallel Computing. Section III deals with introduction of an OpenMP. Section IV deals with related work done in the current field. Section V includes applications of matrix concatenation. Section VI focuses on implementation details. Section VII deals with experimental results, performance evaluation and discussion followed by conclusion in section VIII.

II. PARALLEL COMPUTING

The main aim to 'parallelize' the program code is to reduce the amount of time it takes to run. Consider the time it takes for a program to run to be T, the number of instructions to be executed is I multiplied by the average time (t_{av}) it takes to complete the computation on each instruction:

$$T = I \times t_{av}$$

In this case, it will take a serial program approximately time T to run. If we want to decrease the run time for this program without changing the code, we would need to increase the speed of the processor doing the calculations. However, it is not viable to continue increasing the processor speed indefinitely because the power required running the processor is also increasing. With the increase in power used, there is an equivalent increase in the amount of heat generated by the processor which is much harder for the heat sink to remove at a reasonable speed. As a result of this, we have reached an era where the speeds of processors is not increasing significantly but the number of processors and cores included in a computer is increasing instead [6]. For a parallel program, it will be possible to execute many of these instructions simultaneously. So, in an ideal world, the time to complete the program (Tp) will be the total time to execute all of the instructions in serial (T) divided by the number of processors we are using (Np):

$$Tp = T/Np$$

In reality, programs are rarely able to be run entirely in parallel with some sections still needing to be run in series. Consequently, the real time (Tr) to run a program in parallel will be somewhere in between Tp and T, i.e. $Tp < Tr < T$.

In the 1960's, Gene Amdahl determined the potential speed up of a parallel program, now known as Amdahl's Law. This law states that the maximum speedup of the program is limited by the fraction of the code that can be parallelized:

$$S + P = 1$$

$$SU = 1 / S$$

The serial fraction of the program (S) plus the parallel fraction of the program (P) are always equal to one. The speed-up (SU) is a factor of the original sequential runtime (T). So, if only 50% of the program can be parallelized, the remaining 50% is sequential and the speedup is:

$$SU = 1 / 0.5 = 2$$

i.e., the code will run twice as fast. The number of processors performing the parallel fraction of the work can be introduced into the equation and then the speed-up becomes:

$$SU = 1 / (P/N) + S$$

Where S and P are the serial and parallel fractions respectively and N is the number of processors.

Parallel computing has gained its popularity since 80's after the development of supercomputers with massively parallel architectures. Open MP, Message passing Interface (MPI), Parallel Virtual Machine (PVM), Compute Unified Device Architecture (CUDA), parallel MATLAB etc are several parallel computing platforms available. In the current study, we have selected OMP parallel environment for evaluating the performance of matrix concatenation algorithm.

III. OPENMP

OpenMP is an industry standard application programming interface designed for programming shared memory parallel computers. It uses the concepts of threads and tasks. OpenMP is a set of extensions to FORTRAN, C and C++. The extensions consist of: Compiler directives, Runtime library routines and Environment variables. OpenMP FORTRAN standard released October 1997, minor revision (1.1) in November 1999, Major revision (2.0) in November 2000. OpenMP C/C++ standard released October 1998, Major revision (2.0) in March 2002. OpenMP enable the user to easily expose an application's task and loop level parallelism by providing high level programming constructs in an incremental fashion. OpenMP operates on fork and join model of parallel execution as shown in Fig. 1. It is based upon the existence of multiple threads in the shared memory programming paradigm. A shared memory process consists of multiple threads. OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization [1]. The basic parallel construct in OpenMP is the parallel region. A Parallel Region is a block of code executed by all threads simultaneously. All OpenMP programs begin as a single process, the master thread. An OpenMP program alternates Sequential and parallel regions. The master thread forks into a number of parallel worker threads. The instructions in the parallel region are then executed by the team of worker threads. At the end of the parallel region, the threads synchronize and join to become the single master thread again. Afterwards, they execute the task in parallel using the multiple cores of a processor. OpenMP allows incremental parallelization and very easy to implement on currently existing serial codes [4][6]. OpenMP provides a relaxed-consistency and temporary view of thread memory. Threads and are not required to maintain exact consistency with real memory all of the time and can cache their data. OpenMP allow programmer to write the program that can run faster, if the number of cores are increased and able to use all cores of a multicore computer [4].

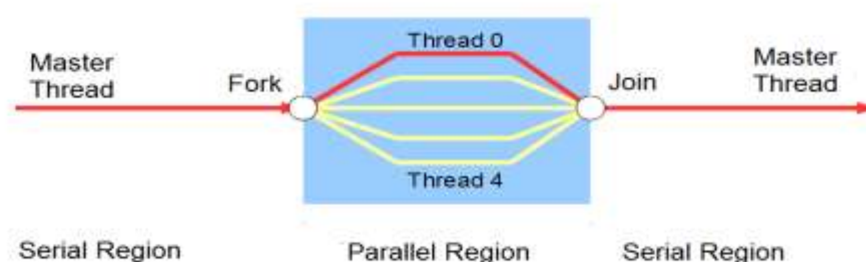

Figure1. Fork and join model

IV. RELATED WORK

In [1], the authors evaluate the performance of matrix multiplication algorithm on dual core 2.0 GHz processor with two threads. The authors implement the algorithm on OpenMP platform by selecting time of execution, speed up and efficiency as performance parameters. Based on the experimental analysis, it was found that a good performance can be achieved by executing the problem in parallel rather than sequential after a certain problem size.

In [2], the authors implemented the bubble sort algorithm using multithreading (OpenMP). The proposed work is tested on two standard datasets (text file) with different size. The main idea of the proposed algorithm is distributing the elements of the input datasets into many additional temporary sub-arrays according to a number of characters in each word. They implement OpenMP using Intel core i7-3610QM, (8 CPUs), using two approaches (vectores of string and array 3D). According to the results the OpenMP shows the best speedup when used 8 threads, the best speedup occurs when using threads number as equal to the actual cores number.

In [3], the pi calculation and Gaussian Elimination algorithms are tested with and without OpenMP and better speedup with parallelization is achieved. In [4], the authors parallelize algorithms on Multi Core systems using OpenMP, results shows significant speed up achieved on multi core systems with the parallel algorithm. In [5], the authors perform analysis and results show that optimizing the code using OpenMP increases the performance than sequential implementation and outperforming well with parallel algorithms. In [6], the authors present 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.

In [7], this paper reviews about OpenMP API and focuses on improving the system performance**.** In [10], the performance (speedup) of parallel algorithms on multi-core system has been presented in this paper. The experimental results on a multi-core processor show that the proposed parallel algorithms achieve good performance compared to the sequential.

V. MATRIX CONCATENATION AND ITS APPLICATION

Matrix concatenation is the process of joining one or more matrices to make a new matrix. One advantage of using matrices is that we can combine the effects of two or more matrices by multiplying them. This means that, to rotate a model and then translate it to some location, we do not need to apply two matrices. Instead, we multiply the rotation and translation matrices to produce a composite matrix that contains all of their effects.

The expression C= [AB] horizontally concatenates matrix A & B while as the expression C= [A; B] vertically concatenates them. Instead of applying each transformation (scaling(S), translation (T) & rotation(R)) individually they can be concatenated into a single transformation matrix for example, a scale, rotation & translation matrix can be combined as:

$$W=S*T*R$$

Transforming a position vector by this concatenation matrix scales, rotates, translates vector in a single operation. However the order of concatenation is important. Concatenation is performed from left to right while matrix multiplication is not commutative. For example suppose we are rendering earth orbiting the sun. It has both a translation away from sun and a rotation around it, and the concatenation should be performed in that order. If wanted to simulate the earth's axial rotation, then must perform that rotation before the orbit transformations. In such a scenario the complete concatenated matrix would appear as:

$$W=R_{axial}*T*R_{orbit}$$

This process, called matrix concatenation, can be written with the following formula:

$$C=M_{1}.M_{2}.M_{n-1}.M_{n}$$

In this formula, C is the composite matrix being created, and M1 through M_n are the individual transformations that matrix C contains. In most cases, only two or three matrices are concatenated, but there is no limit.

VI. IMPLEMENTATION DETAILS

We have implemented both the sequential and parallel algorithms for matrix concatenation. The matrix concatenation sequential algorithm can only perform one computation at a time, whereas the parallel matrix concatenation algorithm is implemented using OMP parallel environment using a multi-core processor. The parallelized program segment is divided into threads and each thread runs independent of other threads. The parallel task is divided into that many threads and each thread runs on an individual core. The main motive behind this approach is to increase the performance (speedup) which is inversely proportional to execution time [6].

The sequential and the parallel code with OpenMP are executed on Intel core2duo processors which have dual cores and each core can execute 2 logical threads. The running time of the algorithm on different processors was noted down and the performance measure (speed up) of the system was evaluated accordingly. The main objective here is to get a better performance as the number of threads increase. An overview of the proposed work is shown in the Fig.2.

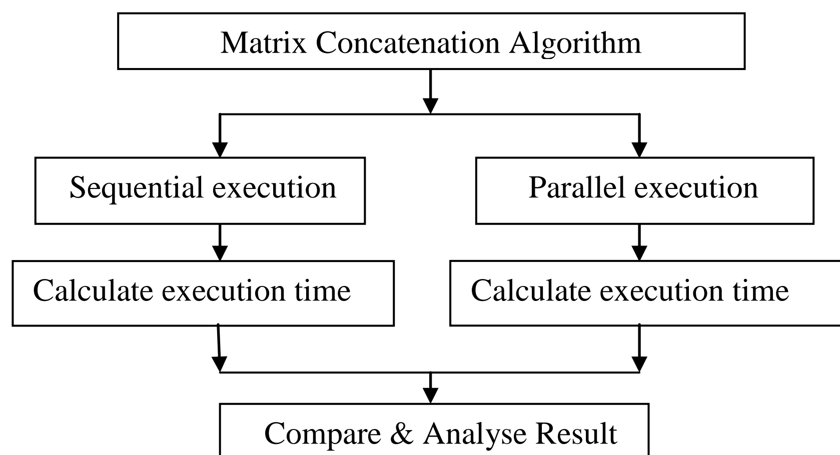

Figure2. Overview of proposed work

Algorithms:

Matrix concatenation without OpenMP

Step 1: Declare variables to store allocated memory

Step 2: Declare variables to input matrix size

Step 3: Declare variable to calculate the time difference between the start and end of the execution.

Step 4: Enter dimension for N*N matrix.

Step 5: Allocate dynamic memory for matrix using malloc function.

Step 6: Initialize first matrix.

```
For (i=0; i<m; i++) {
        For (j=0; j<n; j++) {
             Scanf ("%d", &a[i][j]); }
}
```

Step 7: Initialize second matrix.

```
For (i=0; i<m; i++) {
      For (j=0; j<n; j++) {
            b[i][j]=a[i][j]; }
}
```

Step 8: Start the timer.

```
Start = clock ();
```

Step 9: Do matrix Concatenation

```
For (i=0; i<m; i++) {
      For (j=n; j< (2*n); j++) {
            b[i][j]=a[i][j-n]; }
}
```

Step 10: End the timer.

```
End = omp_get_wtime ();
```

Step 11: Calculate the difference in start and end time.

```
Difference = (end – start) / Clocks_Per_Second.
```

Step 12: Print the resultant concatenated matrix

```
For (i=0; i<m; i++) {
      Printf ("\n");
      For (j=0; j< (2*n); j++) {
           Printf ("%d", b[i][j]); }
}
```

Step 13: Print the time required for program execution without OMP.

Matrix concatenation with OpenMP

Step 1: Declare variables to store allocated memory

Step 2: Declare variables to input matrix size i, j and n.

Step 3: Declare variable to be used by Open MP function for finding the number of threads that can be used in the execution of the program.

Step 4: Declare variable to calculate the starting and ending time for computation.

Step 5: Enter dimension for N*N matrix.

Step 6: Allocate dynamic memory for matrix using malloc function.

Step 7: Initialize first matrix.

```
For (i=0; i<m; i++) {
      For (j=0; j<n; j++) {
            Scanf ("%d", &a[i][j]); }
}
```

Step 8: Initialize second matrix.

```
For (i=0; i<m; i++) {
      For (j=0; j<n; j++) {
            b[i][j]=a[i][j]; }
}
```

Step 9: Start the timer.

```
Start = clock ();
```

Step 10: The Actual Parallel region starts here

```
#pragma omp parallel for private (nthreads, tid) {
      tid = omp_get_thread_num ()
            If (tid == 0) {
             nthreads = omp_get_num_threads ()
             Printf ("Threads %d information\n", tid) ;}
```

Step 11: Do matrix concatenation using parallel pragma directive of OMP.

```
#pragma omp parallel for private (i, j)
      For (i=0; i<m; i++) {
            For (j=n; j< (2*n); j++) {
                  b[i][j]=a[i][j-n]; }
      }
```

Step 12: End the timer.

```
End = omp_get_wtime ();
```

Step 13: Calculate the difference in start and end time.

```
Difference = (end – start) / Clocks_Per_Second.
```

Step 14: Print the resultant concatenated matrix

```
For (i=0; i<m; i++) {
      Printf ("\n");
      For (j=0; j< (2*n); j++) {
            Printf ("%d", b[i][j]); }
}
```

Step 15: Print the time required for program execution with OMP.

VII. EXPERIMENTAL RESULTS

The two algorithms both sequential and parallel are executed on Intel core2duo which is a quad core machine. In order to evaluate the performance of matrix concatenation algorithm in OMP parallel environment, the performance measure i.e., speed up is used. We analyze the result and derive the conclusion.

In OpenMP parallel programming environment, the computational execution times including both sequential and parallel run were recorded. Results were shown in Table 1 and a graph was plotted accordingly in Fig. 3.

Table1. Execution Time of matrix concatenation

Matrix Size	Sequential Program	Parallel Program With Two Threads	Parallel Program With Four Threads
500*500	1.23	1.1	1.08
1000*1000	13.30	9.08	6.43
1500*1500	53.45	28.23	20.19
2000*2000	129.17	71.30	50.60
2500*2500	279.31	148.63	109.05
3000*3000	455.42	234.48	188.07
3500*3500	827.67	440.87	308.16
4000*4000	1169.76	627.04	508.90
4500*4500	1549.34	931.24	718.25

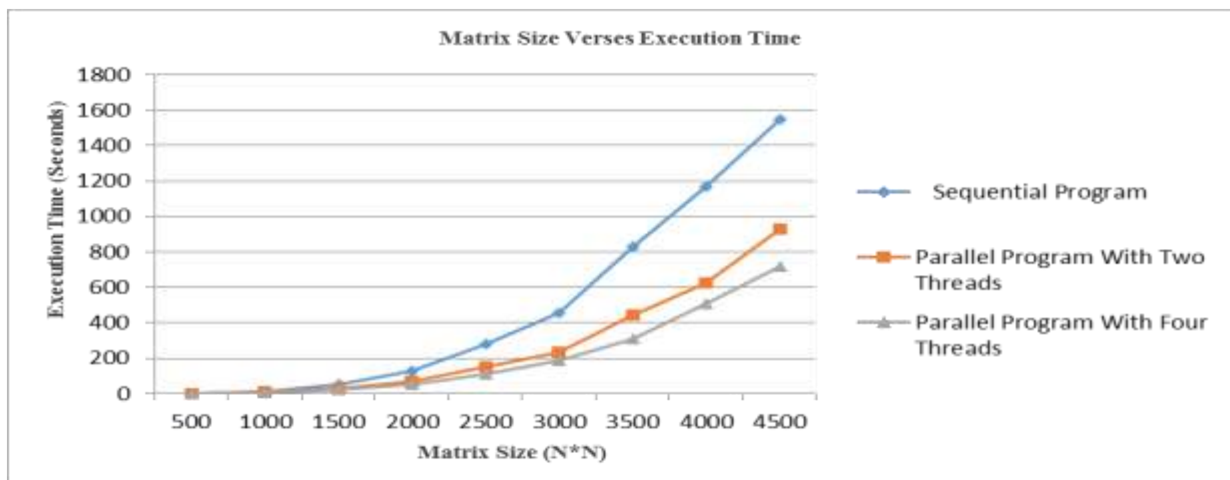

Figure3. Execution time of sequential and parallel algorithm of matrix concatenation

To analyze the performance of matrix concatenation algorithm in OpenMP programming platform, speedup is calculated. These results were presented in Table 2 and Fig. 4.

Table2. Speedup for matrix concatenation

Matrix Size	Two Threads	Four Threads
500*500	1.11	1.13
1000*1000	1.46	2.06
1500*1500	1.89	2.64
2000*2000	1.81	2.55
2500*2500	1.87	2.56
3000*3000	1.94	2.42
3500*3500	1.87	2.68
4000*4000	1.86	2.29
4500*4500	1.66	2.15

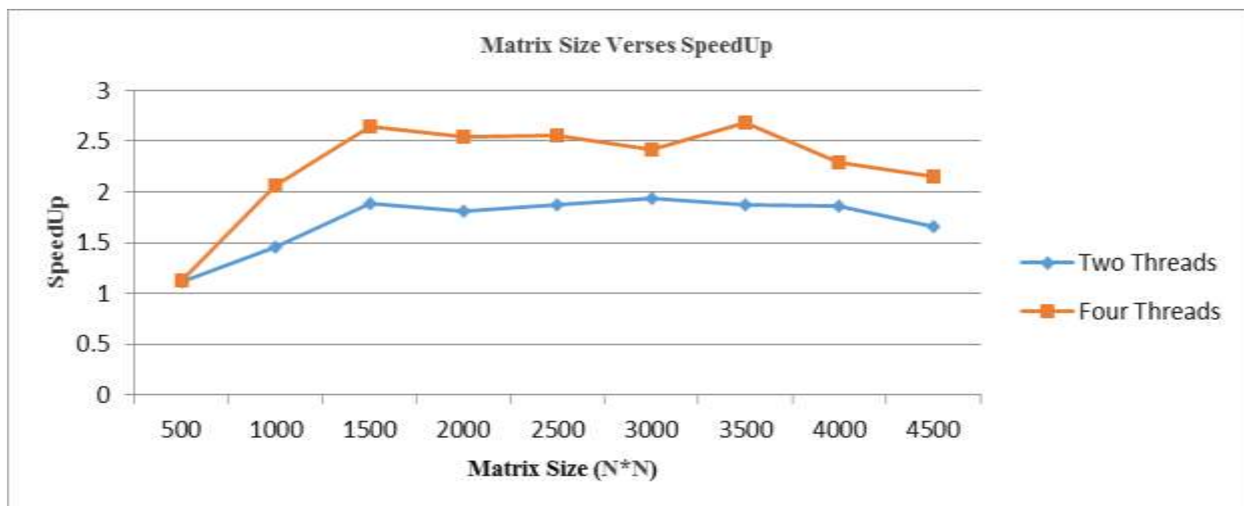

Figure4. Speedup graph for matrix concatenation

VIII. CONCLUSION

OpenMP is beneficial for multi core programming and a great tool for parallel computing environment having richness of functionalities. OpenMP provides a lot of performance increase and parallelization for multi core systems. As the number of cores increase, the computation time taken by an algorithm is also less. When data set size is small the parallel algorithm cannot perform better than sequential algorithm, as the size of the data set increases the execution of parallel algorithm starts performing better and provides much better outcomes than the sequential execution. The system performance can further be improved by lots of parallelization. OpenMP is a very good parallel platform to achieve high performance.

REFERENCES

[1]Y. Dash, S. Kumar, V.K. Patle, Evaluation of Performance on Open MP Parallel Platform based on Problem Size, *I.J. Modern Education and Computer Science, Vol.6,* 2016**,** 35-40.

[2] Z.A.A. Alyasseri, K. Al-Attar, M. Nasser, Parallelize Bubble Sort Algorithm Using OpenMP , *International Journal of Advanced Research in Computer Science and Software Engineering (IJARCSSE), Vol.4, Issue 1*, 2014.

[3] J. Breckling, S.K. Sharma, K. 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*, 2012.

[4]S. Kathavate, N.K. Srinath, Efficient of Parallel Algorithms on Multi Core Systems Using OpenMP, *International Journal of Advanced Research in Computer and Communication Engineering, Vol. 3, No. 10*, 2014, 8237-8241.

[5] V. Saravanan, M. Radhakrishnan, A.S. Basavesh, and D.P. Kothari, A Comparative Study on Performance Benefits of Multi-core CPUs using OpenMP, *IJCSI International Journal of Computer Science Issues, Vol. 9, Issue 1, No 2*, 2012.

[6]M. Basthikodi, W. Ahmed, Parallel Algorithm Performance Analysis using OpenMP for Multicore Machines, *International Journal of Advanced Computer Technology (IJACT), Vol. 4, No 5*, 28-32.

[7] Ms. Ashwini, M. Bhugul, Parallel Computing using OpenMP, *International Journal of Computer Science and Mobile Computing (IJCSMC), Vol. 6, Issue.*2, 2017, 90 – 94**.**

[8] A.C. Sodan, J. Machina, A. Deshmeh, K. Macnaughton, Parallelism via multithreaded and Multicore CPUs, *IEEE Computer Society, Vol. 43, Issue.*3, 2010, 24-32.

[9] P. Graham, Edinburgh, A Parallel Programming Model for Shared Memory Architectures, *Parallel Computing Center, The University of Edinburgh*, 2011.

[10] P. Kulkarni, S. Pathare, Performance analysis of parallel algorithm over sequential using OpenMP, *IOSR Journal of Computer Engineering (IOSR-JCE), Vol. 16, Issue.*2, 2014, 58-62.

[11] J. Ali, R.Z. Khan, Performance Analysis of Matrix Multiplication Algorithms Using MPI, *International Journal of Computer Science and Information Technologies (IJCSIT), vol. 3, No.1*, 2012, 3103 -3106.