A Comparative Study on Performance Benefits of Multi-core CPUs using OpenMP

Vijayalakshmi Saravanan[1], Mohan Radhakrishnan [2], A.S.Basavesh [2], and D.P. Kothari[3]

Ryerson University, Canada[1], HCL Canada, Canada[2], NITK, India[2] , VITS, Nagpur[3]

Abstract
Achieving scalable parallelism from general programs was not successful to this point. To extract parallelism from programs has become the key focus of interest on multi-core CPUs. There are many techniques and programming models such as MPI, CUDA and OpenMP adopted in order to exploit more performance. But there is an urge to find the best parallel programming techniques for the benefit of performance. This article shows how the performance potential benefits the parallel programming model over sequential programming model. To support our claim, we are likely to analyze the performance in terms of execution time on both sequential and parallel implementations of naive matrix multiplication vs. Strassen's matrix multiplication algorithm using OpenMP. Our analysis results show that optimizing the code using OpenMP increases the performance than sequential implementation and outperforming well with parallel algorithms.
Keywords: *Multi-core, Performance Analysis, OpenMP, Strassen's Algorithm, Parallelism.*

1. Introduction

In the recent years, the computer architects no longer rely on increasing single-core processor clock speed or micro architectural improvements to enhance processor performance and found it difficult in exploiting more instruction level parallelism from a single program. Thread-level parallelism could be a well-known strategy to improve processor performance. So, this results in multithreaded processors. Unfortunately, most applications are not multithreaded. Thus, adding cores results in little performance improvement. Researchers have proposed many programming languages to exploit parallelism [1] [5] [6]. These languages allowing high-level parallelism makes parallel programming easier than earlier methods.

Matrix multiplication is an important core computation in many areas of scientific computing. Normally for small multiplication we lean towards to use naive matrix multiplication algorithm which has rich data parallelism. To obtain more performance through that algorithm we parallelized them using OpenMP.

As matrix size grows the naive matrix multiplication becomes inefficient in terms of performance. For large matrices, we used Strassen's algorithm for matrix multiplication (recursive, divide and conquer approach), to enhance the performance of this algorithm on multi core architecture which has functional parallelism in its algorithm, we used OpenMP to parallelize. The results of using OpenMP in each algorithm were encouraging.

The rest of the paper as follows Section 2 describes about overview of OpenMP. Section 3. Brief about related work. Section 4 describes the algorithm and implementation methodology. Section 5 explains the tabulation of how OpenMP helps to improve performance of multi-core processors using Strassen's vs. naive algorithm. Section 6 discusses Result analysis and Discussions. Section 7 finally provides the conclusion and future work.

Hardware and Software Used:

Table 1: System specifications

Processor		Intel Core i3 Dual-core
CPU		2.13GHz
RAM		4GB
Operating System		Windows 7/ Ubuntu 9.04 or later
Soft wares		Visual Studio 2005, GCC compiler (Linux)

2. Overview of an OpenMP

The OpenMP (open multiprocessing) is an application interface platform for shared memory and it consists of set of compiler directives, library routines and environment variables that directs run time behavior. Multiprocessor programming in C or C++ on such architecture includes UNIX and windows operating systems [9]. Due to the boom in hardware speeds and the drop in hardware costs, several developers have let code optimization slip to the back of their minds. As a result, the previously developed techniques from years ago have not been updated to account for modern compiler optimization techniques or hardware features. Synthesizing a large volume of data from opposing viewpoints led to the development of a general outline to follow when optimizing code.

Many programmers will choose a language they are familiar with, even though if it's not the most effective language for the research work. Speed, flexibility, and ease of coding are a few of the major factors in deciding which language to use. The compiler will perform several optimizations faster than human programmer does. Optimization like moving constant expressions outside of loops, storing variables in registers, moving functions inline, and unrolling loops should be performed by the compiler in most cases. Parallelization of sequential programs, parallelizing compiler depend upon subscript analysis to detect data dependencies between pairs of array references inside the loop nests.

To understand the concept of OpenMP, there's a necessary to understand the concept of parallel programming. Parallel processing is done by more than one processor in parallel computing systems. Earlier multiprocessing systems always came in its own processor packaging, however recently introduced multi-core processor can contain multiple processor or cores on a single chip. In this work we achieved thread level parallelism using OpenMP and it reduces the communication cost. OpenMP is an API which acts as parallel programming model on multi-core architecture.

3. Related work

Prior work has studied the implementations of Strassen's matrix multiplication algorithms in many programming languages such as C, C++ and Java [4]. But there is a need to understand the parallel programming and its implementation methodologies on multi processors system in order to improve the performance. OpenMP is the well-known parallel programming techniques for multiprocessing environment [2]. There are varied hardware and software techniques adopted for performance enhancements.

One of the traditional methods to achieve more performance is to increase the clock frequency. There are different kinds of heterogeneous pipeline models have been discussed by many researchers. Latch based pipelines are most commonly used pipelines in asynchronous circuit pipeline models [10]. Fine grained and coarse grained pipeline structure which focuses on cell gate implementation were introduced and improved by many computer architecture researchers [7] [3].

Recently, SR (Self Resetting) latches were proposed by [8] which resolve the power consumption problem and reduces the data path power consumption. Kunkel and Smith studied the performance improvement using gate level logic circuits. As there's tremendous improvement in silicon technology the problems of clocking, range of transistors on chip, and will increase the complexity on chip. Therefore there's an urge to find the software or hardware algorithm to solve this issue.

4. Algorithm and Implementation Methodology

(A) Sequential naive algorithm

We used both sequential and parallel versions of naive and Strassen's algorithm to analyze the performance shown in Figure.6. The pseudo code for the naive algorithm of matrix multiplication of matrix a (n*n) and matrix b (n*n) to give a matrix c (n*n) is shown in Figure.1 [14].

(B) Parallel naive algorithm using OpenMP

The pseudo code for the naive algorithm of matrix multiplication of matrix a (n*n) and matrix b (n*n) to give a matrix c (n*n) is shown in Figure.2 [14].
It can be viewed as divide and conquer method algorithm, suppose we wish to compute the product of

$$C = A * B \text{ ------ } (1)$$

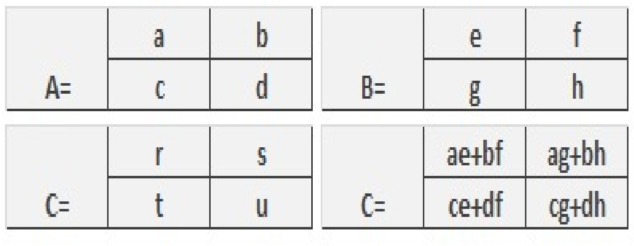
```
SQUARE-MATRIX-MULTIPLY (A, B, C)
N = A.rows
For I = 1 to N
For j = 1 to N
$C_{ij} = 0$
For k = 1 to N
$C_{ij} = C_{ij} + A_{ik} * B_{kj}$
Return C
```

Fig. 1: Pseudo code for sequential naive algorithm

```
SQUARE-MATRIX-MULTIPLY (A, B, C)
#pragma omp parallel for default (none) shared (A, B, C, N) private (I, j)
N = A.rows
For I = 1 to N
For j = 1 to N
$C_{ij} = 0$
For k = 1 to N
$C_{ij} = C_{ij} + A_{ik} * B_{kj}$
Return C
```

Fig. 2: Pseudo code for parallel naive algorithm

In Equation [1], where each of A, B and C are n*n matrices. Assuming that n is an exact power of two, we tend to divide each of A, B and C into four n/2*n/2 matrices, which can be written as shown in Figure.3.



	a	b		e	f
A=	c	d	B=	g	h
	r	s		ae+bf	ag+bh
C=	t	u	C=	ce+df	cg+dh

Fig. 3: Strassen’s serial algorithm

4.1 Strassen’s Serial Algorithm

As we can see from the above, the serial algorithm has recursion in the algorithm. We can hardly see the data parallelism except in adding and subtracting sub matrices. If the matrix size is greater than the threshold value multiply them recursively, if not use the traditional matrix multiplication algorithm.

Construct C using the intermediate matrices. But, if we look more as shown in Figure.4 and we get to understand that P1….P7 goes on recursively and independently thus we get functional parallelism.

4.2 Strassen’s Parallel Algorithm

Initially, we implemented our program through task pool model to compute P1; P2...P7 and an independent multiplication task can be executed in parallel with N jobs at a time. For example, when 49 jobs are running with N cores machine (where N= 2^N) there is a chance to execute 48, and would run simultaneously with 1 job would be left later execution. Thus, it leads to more processor utilization. So as to avoid this issue, it’s better to split the last task further as shown in Figure 7.

In OpenMP the sections construct is the easiest way to get the different threads to carry out different kinds of work. Since, it permits us to specify many different code regions and each of which will be executed by one of the threads in OpenMP with Strassen’s matrix multiplication are shown in Figures 4 and 5 [12] [13].

Fig.4 STRASSEN’S MATRIX MULTIPLICATION

$$A = \begin{matrix} A_{11} & A_{12} \\ A_{21} & A_{21} \end{matrix}$$

$$B = \begin{matrix} B_{11} & B_{12} \\ B_{21} & B_{21} \end{matrix}$$

$$C = \begin{matrix} C_{11} & C_{12} \\ C_{21} & C_{21} \end{matrix}$$

$$C = \begin{matrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{matrix} * \begin{matrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{matrix}$$

$C_{11} = A_{11}*B_{11} + A_{12}*B_{21}$

$C_{12} = A_{11}*B_{12} + A_{12}*B_{22}$

$C_{21} = A_{21}*B_{11} + A_{22}*B_{21}$

$C_{22} = A_{21}*B_{12} + A_{22}*B_{22}$

Fig.5 Strassen's Parallel Algorithm

Evaluate the intermediate matrices:

$P_1 = (A_{11} + A_{22}) (B_{11} + B_{22})$

$P_2 = (A_{21} + A_{22}) B_{11}$

$P_3 = A_{11} (B_{12} – B_{22})$

$P_4 = A_{22} (B_{21} – B_{11})$

$P_5 = (A_{11} + A_{12}) B_{22}$

$P_6 = (A_{21} – A_{11}) (B_{11} + B_{12})$

$P_7 = (A_{12} – A_{22}) (B_{21} + B_{22})$

Construct C using the intermediate matrices:

$C_{11} = P_1 + P_4 – P_5 + P_7$

$C_{12} = P_3 + P_5$

$C_{21} = P_2 + P_4$

$C_{22} = P_1 – P_2 + P_3 + P_6$

5. Tables for Performance Analysis of Strassen's vs. Naïve Algorithm

Table 2.Tabulation of performance analysis on Strassen's vs. Naïve algorithm

Matrix Size (n)	Naive Serial	Naive Parallel	Strassen's Serial	Strassen's Parallel
500	1.23	0.88	2.4	2.2
1000	13.2	8.11	6.2	5.4
1500	59.35	31.92	12.6	7.8
2000	99.62	79	22.3	10.81
2500	279.23	178.5	40.23	19.46
3000	394.5	316.62	62.35	28.86

In our work, we have tested the sequential version and parallel version (using OpenMP) for both naive and Strassen's algorithm for matrix multiplication. The execution time was taken using OpenMP run time library function omp_get_wtime () which gives time in seconds with double precision. Using OpenMP the parallelism can be achieved through the evaluations of intermediate matrices P1, P2 ... P7 which are independent as shown in Figure.5 and hence, it will be computed in parallel through Strassen's parallel matrix multiplication. Comparing the serial and parallel version of naive algorithm we got significant results from matrix sizes of more than 100*100 due to time consumed in thread synchronizing for smaller matrices. For Strassen's algorithm initially the performance was quite disappointing for smaller matrices and as the matrix sizes became larger than 500*500 the performance improved compared to naive multiplication and also the parallel version of Strassen's algorithm and the execution time is shown in Table 2 (Time in sec). The results graphs are depicted in Fig.8 and Fig.9.

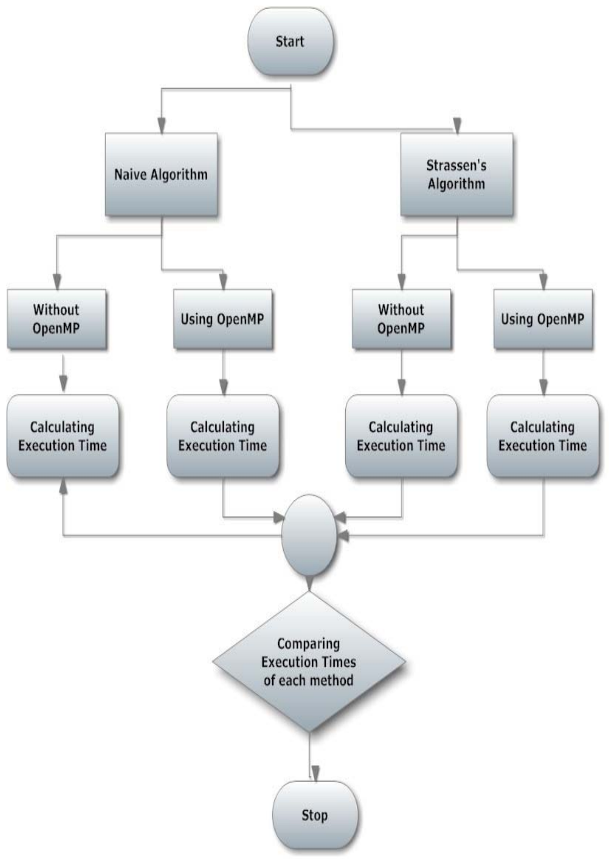

Start
Naive Algorithm
Strassen's Algorithm
Without OpenMP
Using OpenMP
Without OpenMP
Using OpenMP
Calculating Execution Time
Calculating Execution Time
Calculating Execution Time
Calculating Execution Time
Comparing Execution Times of each method
Stop

Fig. 6: Schematic flow diagram

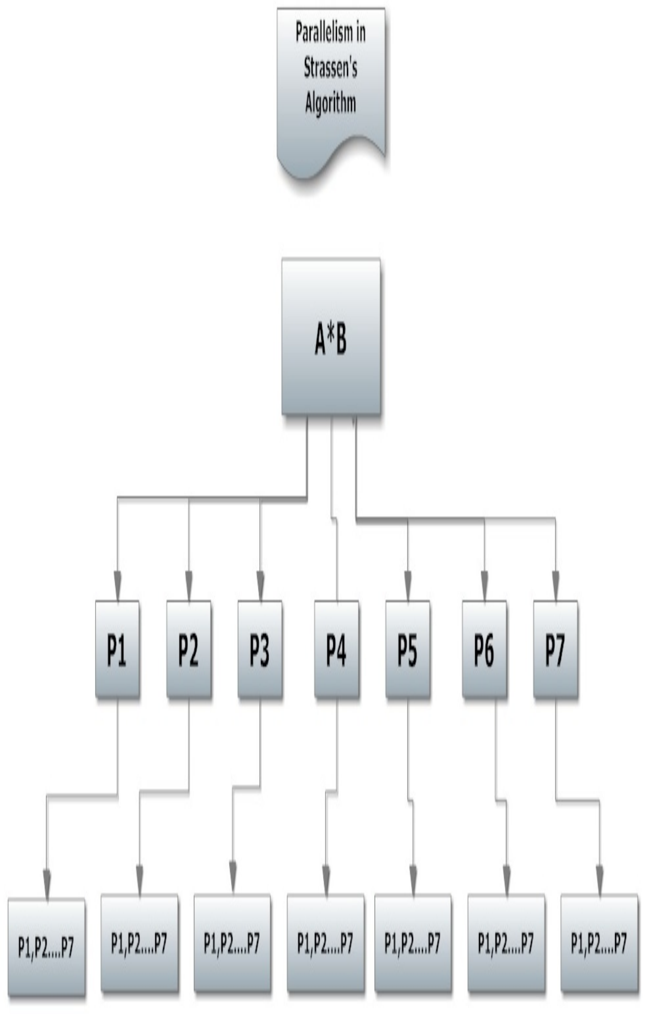


Fig. 7: Flow diagram of Strassen's algorithm

Results Graphs

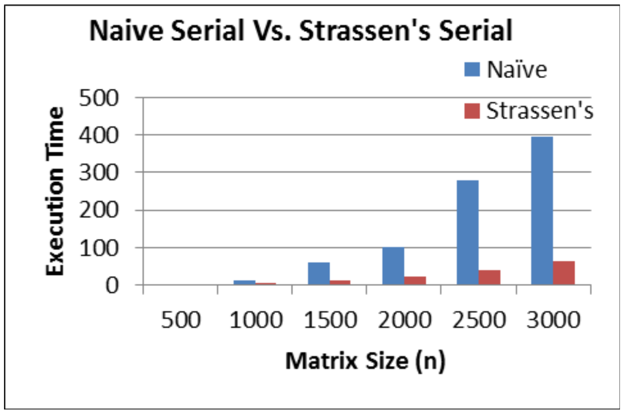


Fig.8 Naive vs. Strassen's Serial Algorithm

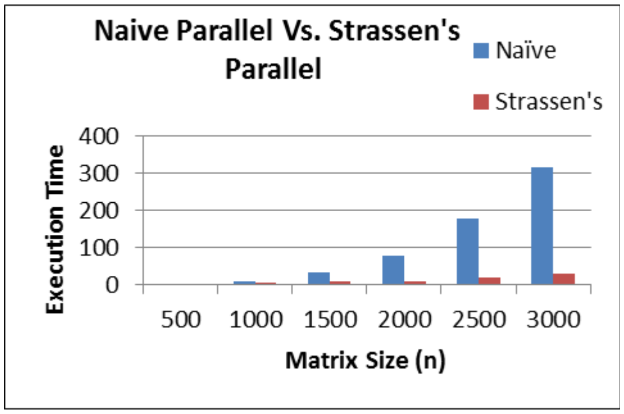


Fig.9 Naïve vs. Strassen's Parallel Algorithm

6. Result analysis and Conclusion

Based on our study, we have presented the execution time of both serial and parallel execution of naive and Strassen's algorithm for matrix multiplication. We arrive at the following conclusions:
(a) we see that parallelizing the serial algorithm using OpenMP has increased the performance a lot (b) For CPU cores OpenMP provides a lot of performance increase and parallelization can be done with minimal changes and, (c) we tend to observe that though Strassen's algorithm (both parallel and serial) definitely consumes a lot more memory than serial algorithm, but the performance is much better than the traditional matrix multiplication algorithm due to its reduced operations (d) overall we conclude that for large matrices we can apply Strassen's algorithm and for smaller matrices we must apply naive algorithm. And using OpenMP for both algorithms we achieved much better performance than serial implementation.

7. Future Enhancements

Due to time constraints, this work has been carried out on dual-core machine with matrix multiplication alone, but it can be extended by using a variety of matrix types - dense, sparse, large data, complex numbers, etc. to characterize our comparison to understand the better performance benefits of the OpenMP techniques. Besides there's a scope to look at the energy consumption of assorted algorithms and its impact on performance enhancement.

Acknowledgments

The authors would like to convey immense thankfulness to IASc (Indian Academy of Sciences, Bangalore) and all the anonymous reviewers for their valuable comments.

References

[1] Brian D et al. Carlstrom.”The Atomos Transactional Programming Language”. In ACMSIGPLAN2006 Conference on Programming Language Design and Implementation. June 2006.
[2] Barbara Chapman. ”Managing Multi-core with OpenMP (Extended Abstract)”. In Proceedings of the 15th European PVM/MPI Users’ Group Meeting on “Recent Advances in Parallel Virtual Machine and Message Passing Interface”, pages 3–4, Berlin, Heidelberg, 2008. Springer Verlag.
[3] Zenil Chavez. ”Applied Parallel Computing”. IEEE Distributed Systems Online, 5, 2004.
[4] Thomas H.et al. Cormen. ”Introduction to Algorithms”. McGraw-Hill Higher Education, 2nd edition, 2001.
[5] Matteo et al. Frigo. ”The implementation of the Cilk 5 multithreaded language”. In Proceedings of the ACMSIGPLAN1998 conference on Programming language design and implementation, PLDI ’98, pages 212–223, New York, NY, USA, 1998. ACM.
[6] Michael I. et al. Gordon. ”A stream compiler for communication exposed architectures”. SIGARCH Comput.Archit. News, 30:291–303, October 2002.
[7] Shi Jung Kao. ”Managing C++ OpenMP code and its exception handling”. In Proceedings of the OpenMP applications and tools 2003 international conference on OpenMP shared memory parallel programming, WOMPAT’03, pages 227–243, Berlin, Heidelberg, 2003. Springer-Verlag.
[8] Quin Michael J.”Parallel programming in C with MPI and OpenMP”. McGraw Hill Inc., 2004.
[9] Venkatesan Packirisamy, Harish Barathvajasankar, S Sarholz in Proceedings of the 3rd international workshop on OpenMP "A Practical Programming Model for the Multi-core Era" (2008).
[10] Alex Vrenios. ”A Tutorial on Parallel Systems Development”. IEEE Distributed Systems Online, 5, 2004.
[11] www.OpenMP.org.
[12]http://ace.cs.ohiou.edu/~razvan/courses/cs404/lecture12.pdf.
[13] Steven Huss-lederman, Elaine M. Jacobson, J. R. Johnson, Anna Tsao, Thomas Turnbull “Strassen’s Algorithm for Matrix Multiplication: Modeling, Analysis, and Implementation” In Proceedings of Supercomputing '96.
[14] John Burckardt, Paul Puglielli Pittsburgh Supercomputing Center, “MATMUL: An Interactive Matrix Multiplication Benchmark “.

Vijayalakshmi Saravanan is an Assistant Professor (Sr); VIT University, India .She is a recipient of Erasmus Mundus (EURECA) Programme as an Exchange student from India at Malardalen University, Sweden. She holds a Bachelor of Engineering Degree in Electrical and Electronics Engineering and Master of Science Degree in Information Technology from Bharathiar University & Manonmaniam Sundaranar University (Now Anna University), India. Currently, she holds a position as visiting researcher at Ryerson University, Canada. Her research interests include Multi-core Low Power Design Exploration, Power-Aware Processor Design, and Computer Architecture. She has taken part of her research studies one course work at University of Rochester, USA. She is serving as a Technical Evangelist for Asia Open Source Software Community, CICC, and Japan and all over Asian Countries. She is a Member of IEEE, ACM, CSI and a Board member of N2WOMEN (Networking Networking Women) IEEE/ACM Women in Engineer and she is a Chair for IEEE-WIE VIT affinity group, India. She can be reached at viji@ieee.org.

Mohan Radhakrishnan is currently working as a Sr.Technical Architect in HCL Canada. He has more than ten years of technical experience in designing, administrating and supporting Microsoft enterprise and VMware environments. He is currently working on R&D level projects in data center server and network implementation, support and administration, thorough grasp of development principles and best practices. He is also a Member of IEEE and VMware.

Dr. D.P. KOTHARI is a Senior Professor and Advisor to the Chancellor, VIT University, Vellore and named IEEE fellow in 2011. Earlier, he was Head, Centre for Energy Studies, IIT Delhi (1995-97), and Principal, Visvesvaraya Regional Engineering College, Nagpur (1997-98). He has been Director i/c, IIT Delhi (2005) and Deputy Director (Administration) (2003-06). Earlier, (1982-83 and 1989), he was a visiting fellow at RMIT, Melbourne, Australia. He obtained BE, ME and PhD degrees from BITS, Pilani. He is a Fellow of the Institution of Engineers (India), Fellow of National Academy of Engineering (FNAE), Fellow of National Academy of Sciences (FNASc), Life Member ISTE (LMISTE). Professor Kothari has published/presented 640 papers in national and international journals/conferences. He has authored/co-authored 22 books including Power System Optimization, Modern Power System Analysis, Electric Machines, Power System Transients, Theory and Problems of Electric Machines, Renewable Energy Sources and Emerging Technologies, and Power System Engineering. His research interests include Optimal Hydro-thermal Scheduling, Unit Commitment, Maintenance Scheduling, Energy Conservation (loss minimization and voltage control), and Power Quality and Energy Systems Planning and Modeling. He has received the National Khosla award for Lifetime Achievements in Engineering for 2005 from IIT Roorkee. The University Grants Commission (UGC) has bestowed UGC National Swami Pranavananda Saraswati award for 2005 on Education for outstanding scholarly contribution. The World management congress, New Delhi conferred Life time achievement award for "Educational Planning and Administration" on 30th December 2009.