Concurrent Matrix Multiplication on Multi-Core Processors

Muhammad Ali Ismail *maismail@neduet.edu.pk*
Assistant Professor, Faculty of Electrical & Computer Engineering
Department of Computer & Information Systems Engineering
NED University of Engineering & Technology
Karachi, 75270, Pakistan

Dr. S. H. Mirza *shmirza@uit.edu*
Professor
Usman Institute of Technology
Karachi, 75300, Pakistan

Dr. Talat Altaf *deanece@neduet.edu.pk*
Professor, Faculty of Electrical & Computer Engineering
Department of Electrical Engineering
NED University of Engineering & Technology
Karachi, 75270, Pakistan

Abstract

With the advent of multi-cores every processor has built-in parallel computational power and that can only be fully utilized only if the program in execution is written accordingly. This study is a part of an on-going research for designing of a new parallel programming model for multi-core architectures. In this paper we have presented a simple, highly efficient and scalable implementation of a common matrix multiplication algorithm using a newly developed parallel programming model SPC^3 PM for general purpose multi-core processors. From our study it is found that matrix multiplication done concurrently on multi-cores using SPC^3 PM requires much less execution time than that required using the present standard parallel programming environments like OpenMP. Our approach also shows scalability, better and uniform speedup and better utilization of available cores than that the algorithm written using standard OpenMP or similar parallel programming tools. We have tested our approach for up to 24 cores with different matrices size varying from 100 x 100 to 10000 x 10000 elements. And for all these tests our proposed approach has shown much improved performance and scalability.

Keywords: Multi-Core, Concurrent Programming, Parallel Programming, Matrix Multiplication.

1. INTRODUCTION

Multi-core processors are becoming common and they have built-in parallel computational power and which can only be fully utilized only if the program in execution is written accordingly. Writing an efficient and scalable parallel program is much complex. Scalability embodies the concept that a programmer should be able to get benefits in performance as the number of processor cores increases. Most software today is grossly inefficient, because it is not written with sufficient parallelism in mind. Breaking up an application into a few tasks is not a long-term solution. In order to make most of multi-core processors, either, lots and lots of parallelism are actually needed for efficient execution of a program on larger number of cores, or secondly, concurrent execution of multiple programs on multiple cores [1, 2].

Matrix Multiplication is used as building block in many of applications covering nearly all subject areas. Like physics makes use of matrices in various domains, for example in geometrical optics and matrix mechanics; the latter led to studying in more detail matrices with an infinite number of rows and columns. Graph theory uses matrices to keep track of distances between pairs of vertices in a graph. Computer graphics uses matrices to project 3-dimensional space onto a 2-dimensional screen. Matrix calculus generalizes classical analytical concept such as derivatives of functions or exponentials to matrices etc [4, 11, 13]. Serial and parallel matrix multiplication is always be a challenging task for the programmers because of its extensive computation and memory requirement, standard test set and broad use in all types of scientific and desktop applications. With the advent of multi-core processors, it has become more challenging. Now all the processors have built-in parallel computational capacity in form of cores and existing serial and parallel matrix multiplication techniques have to be revisited to fully utilize the available cores and to get the maximum efficiency and the minimum executing time [2, 3, 8, 9].

In this paper we have presented a concurrent matrix multiplication algorithm and its design using a new parallel programming model SPC^3 PM, (**S**erial, **P**arallel, and **C**oncurrent **C**ore to **C**ore **P**rogramming **M**odel) developed for multi-core processors. It is a serial-like task-oriented multi-threaded parallel programming model for multi-core processors that enables developers to easily write a new parallel code or convert an existing code written for a single processor. The programmer can scale it for use with specified number of cores. And ensure efficient task load balancing among the cores.

The rest of the paper is organized as follows. In section 2, the related studies on parallel and concurrent matrix multiplication are briefly reviewed. The characteristics of SPC^3 PM are described in section 3. Section 4 deals with the programming in SPC^3 PM. The concurrent matrix multiplication algorithm based on SPC^{3}PM is presented in section 5. In section 6 and 7, the experimental setup and results are discussed respectively. Finally, conclusion and future work are given in section 8.

2. RELATED WORK

Many of parallel matrix multiplication algorithms and implementations for SMPs and distributed systems have been proposed. Like Systolic algorithm [5], Cannon's algorithm [], Fox's algorithm with square decomposition, Fox's algorithm with scattered decomposition [6], SUMMA [7], DIMMA [10], 3-D matrix multiplication [12] etc. Majority of the parallel implementations of matrix multiplication for SMPs are based on functional parallelism. The existing algorithms for SMPs are not so efficient for multi-core and have to be re-written using some multi-core supported language [1, 2]. These algorithms are also difficult for common programmer to understand as they require detailed related subject knowledge. On the other hand distributed algorithms which are usually base on data parallelism also cannot be applied on the shared memory multi-core processors because of the architectural change.

Some attempts have also been made to solve matrix multiplication using data parallel or concurrent approaches on cell or GPUs [14, 15, 16, 17, 18, 19]. But the associated problem with these approaches is architectural dependence and cannot be used for general purpose multi-core processors.

3. SPC^3 PM

SPC^3 PM, (**S**erial, **P**arallel, **C**oncurrent **C**ore to **C**ore **P**rogramming **M**odel), is a serial-like task-oriented multi-threaded parallel programming model for multi-core processors, that enables developers to easily write a new parallel code or convert an existing code written for a single processor. The programmer can scale it for use with specified number of cores. And ensure efficient task load balancing among the cores.

SPC^3 PM is motivated with an understanding that existing general-purpose languages do not provide adequate support for parallel programming. Existing parallel languages are largely targeted to scientific applications. They do not provide adequate support for general purpose multi-core programming whereas SPC^3 PM is developed to equip a common programmer with multi-core programming tool for scientific and general purpose computing. It provides a set of rules for algorithm decomposition and a library of primitives that exploit parallelism and concurrency on multi-core processors. SPC^3 PM helps to create applications that reap the benefits of processors having multiple cores as they become available.

SPC^3 PM provides thread parallelism without the programmers requiring having a detailed knowledge of platform details and threading mechanisms for performance and scalability. It helps programmer to control multi-core processor performance without being a threading expert. To use the library a programmer specifies tasks instead of threads and lets the library map those tasks onto threads and threads onto cores in an efficient manner. As a result, the programmer is able to specify parallelism and concurrency far more conveniently and with better results than using raw threads.. The ability to use SPC^3 PM on virtually any processor or any operating system with any C++ compiler also makes it very flexible.

SPC^3 PM has many unique features that distinguish it with all other existing parallel programming models. It supports both data and functional parallel programming. Additionally, it supports nested parallelism, so one can easily build larger parallel components from smaller parallel components. A program written with SPC^3 PM may be executed in serial, parallel and concurrent fashion. Besides, it also provides processor core interaction to the programmer. Using this feature a programmer may assign any task or a number of tasks to any of the cores or set of cores.

3.1 Key Features

The key features of SPC^3 are summarized below.

- SPC^3 is a new shared programming model developed for multi-core processors.
- SPC^3 PM works in two steps: defines the tasks in an application algorithm and then arranges these tasks on cores for execution in a specified fashion.
- It provides Task based Thread-level parallel processing.
- It helps to exploit all the three programming execution approaches, namely, Serial, Parallel and Concurrent.
- It provides a direct access to a core or cores for maximum utilization of processor.
- It supports major decomposition techniques like Data, Functional and Recursive.
- It is easy to program as it follows C/C++ structure.
- It can be used with other shared memory programming model like OpenMP, TBB etc.
- It is scalable and portable.
- Object oriented approach

4. PROGRAMMING WITH SPC^3 PM

SPC^3 PM provides a higher-level, shared memory, task-based thread parallelism without knowing the platform details and threading mechanisms. This library can be used in simple C / C++ program having tasks defined as per SPC^3 PM Task Decomposition rules. To use the library, you specify tasks, not threads, and let the library map tasks onto threads in an efficient manner. The result is that SPC^3 PM enables you to specify parallelism and concurrency far more conveniently, and with better results, than using raw threads.

Programming with SPC^3 is based on two steps. First describing the tasks as it specified rules and then programming it using SPC^3 library. The figure 1 shows the step by step development of an application using SPC^{3}PM.

4.1 Rules for Task Decomposition

- Identify the parts of the code which can be exploited using Functional, Data or Recursive decomposition
- Defined all those piece of code specified in step 1 as Tasks.
- Identify the loops for the loop parallelism and also defined them as Tasks
- Identify portions of the application algorithm which are independent and can be executed concurrently
- A Task may be coded using either C/C++/VC++/C# as an independent unit.
- Tasks should be named as Task1, Task2,….. Task*N.*
- There are no limits for Tasks.
- Arrange the tasks using SPC^3 library in the main program file according to the program flow.
- A Task may be treated as a function.
- A Task may only intake pointer structure as a parameter. Initialize all the parameters in a structure specific to a Task.
- A structured may be shared or private.
- A Task may or may not return the value. The Task named with suffix 'V' do not return any value. The Task with suffix 'R' do return value.

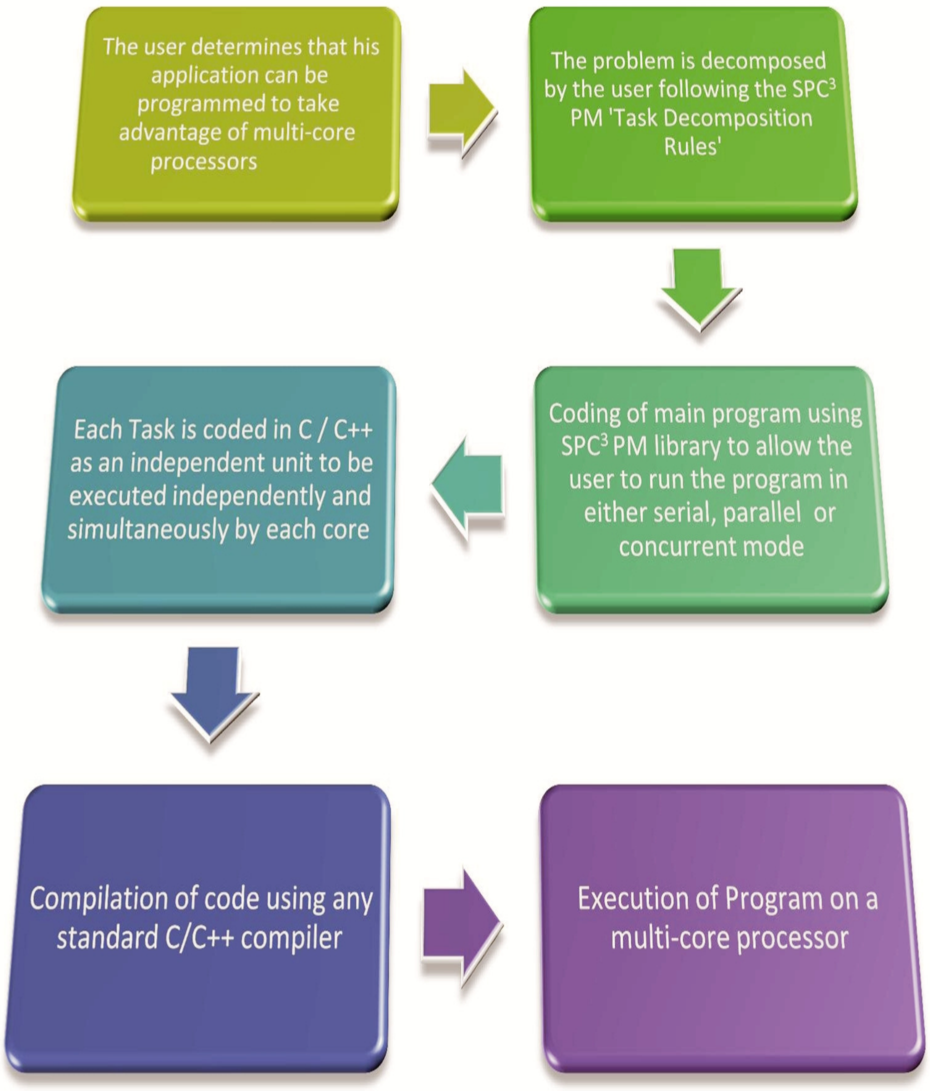

The user determines that his application can be programmed to take advantage of multi-core processors
The problem is decomposed by the user following the SPC3 PM 'Task Decomposition Rules'
Coding of main program using SPC3 PM library to allow the user to run the program in either serial, parallel or concurrent mode
Each Task is coded in C / C++ as an independent unit to be executed independently and simultaneously by each core
Compilation of code using any standard C/C++ compiler
Execution of Program on a multi-core processor

FIGURE 1: Steps involved in programming with SPC^3 PM

4.2 Program Structure

```
Define Task1
Define Task2
Define Task3
Define Task4
...
Define TaskN

Structure             Structure             Structure                                   Structure
STRUCT_NAME           STRUCT_NAME           STRUCT_NAME                                 STRUCT_NAME
{                     {                     {                                           {
//The structure       //The structure       //The structure                             //The structure
//having private      //having private      //having private                             //having private
//or global           //or global           //or global                                 //or global
//parameters          //parameters          //parameters                                //parameters
//associated with a   //associated with a   //associated with a   ………………..   //associated with a
//specified task      //specified task      //specified task                            //specified task
 }                     }                     }                                           }
STRUCT_NAME           STRUCT_NAME           STRUCT_NAME                                 STRUCT_NAME
*P_ TASK1             *P_ TASK2             *P_ TASK3                                   *P_ TASKN

Task1(LPVOID)         Task2(LPVOID)         Task3(LPVOID)                               TaskN(LPVOID)
{                     {                     {                     ………………..   {
//performing          //performing          //performing                                //performing
//some                //some                //some                                      //some
//computation         //computation         //computation                               //computation
}                     }                     }                                           }

void main( void)

{
      // any declaration;
      // any piece of code ;

      Serial (Task1, P_ TASK1 );             //execution of task 1 in serial

      // Any other code ;

      Parallel (Task2, P_ TASK2);            //execution of task 2 in parallel

      // any other code ;

      Concurrent (Task3 , P_ TASK3, Task4,  P_ TA  SK4); //execution of task 3 and 4 concurrently

}
```

4.3 SPC^3 PM Library

SPC^3 PM provides a set of specified rules to decompose the program into tasks and a library to introduce parallelism in the program written using c/ c++. The library provides three basic functions.

- Serial
- Parallel
- Concurrent

Serial: This function is used to specify a Task that should be executed serially. When a Task is executed with in this function, a thread is created to execute the associated task in sequence. The thread is scheduled on the available cores either by operating system or as specified by the programmer. This function has three variants. *Serial (Task i)* {Basic}, *Serial (Task i, core)* {for core specification} and **p Serial (Task i, core, *p)* {for managing the arguments with core specification}

Parallel: This function is used to specify a Task that should be executed in parallel. When a Task is executed with in this function, a team of threads is created to execute the associated task in parallel and has an option to distribute the work of the Task among the threads in a team. These threads are scheduled on the available cores either by operating system or as specified by the programmer. At the end of a parallel function, there is an implied barrier that forces all threads to wait until the work inside the region has been completed. Only the initial thread continues execution after the end of the parallel function. The thread that starts the parallel construct becomes the master of the new team. Each thread in the team is assigned a unique thread id to identify it. They range from zero (for the master thread) up to one less than the number of threads within the team. This function has also four variants. *Parallel (Task i)* {Basic}, *Parallel (Taski ,num-threads)* {for defining max parallel threads}, *Parallel (Task i, core list)* {for core specification} and **p parallel (Task i, core, *p)* {for managing the arguments with core specification}

Concurrent: This function is used to specify the number of independent tasks that should be executed in concurrent fashion on available cores. These may be same tasks with different data set or different tasks. When the Tasks are executed defined in this function, a set of threads equal or greater to the number of tasks defined in concurrent function is created such that each task is associated with a thread or threads. These threads are scheduled on the available cores either by operating system or specified by the programmer. in other words , this function is an extension and fusion of serial and parallel functions. All the independent tasks defined in concurrent functions are executed in parallel where as each thread is being executed either serially or in parallel. This function has also three variants. *Concurrent (Task i, Taskj, ...Task N)* {Basic}, Concurrent (Task i, core , Task j , core, ……) {for core specification} and Concurrent (Task i, core , *p, Task j , core, *p ……) {for managing the arguments with core specification}.

5. CONCURRENT MATRIX ALGORITHM

We have selected a standard and basic matrix multiplication algorithm in which the product of a *(m×p)* matrix *A* with a *(p×n)* matrix *B* is a *(m×n)* matrix denoted *C* such that

$$C_{i,j} = \sum_{k=1}^{p} A_{ik} B_{kj}$$

Where $1 \leq i \leq m$ is the row index and $1 \leq j \leq n$ is the column index. This algorithm is implemented using two different approaches. The first is the standard parallel approach using OpenMP. The other is in C++ using the concurrent function of SPC^3 PM. Pseudo code for both of the algorithms are shown in table 1.

In OpenMP implementation the basic computations of addition and multiplication are placed within the three nested 'for' loops. The outer most is parallelized using OpenMP keyword 'pragma omp parallel for'. The row level distribution of matrices is followed. The matrix is divided into set of rows equal number of parallel threads defined by the variable 'core 'such that each row set is computed on a single core.

For SPC^3 PM using concurrent function, a Task is defined having the basic algorithm implementation. The idea is to execute this task concurrently on different cores with different data set. Every Task has its own private data variables defined in a structure ‘My_Data’. All the private structures are associated with their tasks and initialized accordingly. Using the Concurrent function of SPC^3 PM, the required number of concurrent tasks are initialized and executed.

Matrix Multiplication Algorithm
OpenMP (Parallel)

```
Void main (void)
{

// inintilizillig the matrics
int A[ ][ ],B[ ][ ],C[ ][ ]

int core ; // number of parallel threads

omp_set_num_threads(core);

// initializing the parallel loop
#pragma omp parallel for private(i,j,k)

for (i=0; i<n; i++)
 {
 for (j=0; j<n; j++)
   {
    c[i][j]=0;
    for (k=0;k<n;k++)
      {
      c[i][j]=c[i][j]+ a[i][k]*b[k][j];
      }
   }
 }

}
```

Matrix Multiplication Algorithm
SPC^3 PM, Concurrent

```
Task(LPVOID)
{
P_MY_DATA data;
data=(P_MY_DATA)lp;

for(i=data->val3; i<data->val1; i++)
for(j=0; j< data->val2; j++)
{
for(k=0;k< data->val2 ;k++)
c[i][j]=c[i][j]+ a[i][k]*b[k][j];
}
}

void main (void)
{

 typedef struct My_Data
{
int val1,val2,val3;
int A[ ][ ],B[ ][ ],C[ ][ ]
} MY_DATA, *P_MY_DATA[n];

//initialize P_MY_DATA_1;
//initialize P_MY_DATA_2;
......

//initialize P_MYDATA_N;

concurrent(Task,P_MYDATA_1,Task,P_MY_DATA_2
          ....      Task,P_MY_DATA_N);
}
```

TABLE 1: Parallel Matrix Algorithm for OpenMP and SPC^3 Concurrent

6. EXPERIMENTAL SETUP

For the execution of the algorithms we used quad Intel Xeon processor 5500 series based SR1670HV, server systems having 48 cores and dual Intel Xeon processors 5500 series based SR1600UR server systems with 24 cores. Operating systems used are windows server 2003 and 2008. We tested our approach for up to 24 cores with different matrices size varying from 100 x 100 to 10000 x 10000 elements.

7. PERFORMANCE EVOLUTION

The following tables 2 to 5 show the execution time in seconds for each of two approaches, OpenMP and SPC^3 PM Concurrent with different sizes of matrices for 4, 8, 12 and 24 parallel / concurrent threads respectively.

Matrix Size	Number of Parallel Threads	Execution Time (Sec)	
		OpenMP (Parallel)	SPC^3 PM, Concurrent
100 X 100	4	1	1
1000 X 1000		3	3
2000 X 2000		36	23
3000 X 3000		162	85
4000 X 4000		404	202
5000 X 5000		738	396
6000 X 6000		1244	682
7000 X 7000		2078	1086
8000 X 8000		3093	1619
9000 X 9000		4558	2303
10000 X 10000		5425	3161

TABLE 2: Execution Time (Sec) for parallel matrix multiplication using OpenMP and SPC^3 PM Concurrent for 4 parallel threads

Matrix Size	Number of Parallel Threads	Execution Time (Sec)	
		OpenMP (Parallel)	SPC^3 PM, Concurrent
100 X 100	8	1	1
1000 X 1000		2	1
2000 X 2000		25	12
3000 X 3000		83	44
4000 X 4000		212	104
5000 X 5000		433	204
6000 X 6000		703	351
7000 X 7000		1099	559
8000 X 8000		1742	833
9000 X 9000		2503	1186
10000 X 10000		3276	1626

TABLE 3: Execution Time (Sec) for parallel matrix multiplication using OpenMP and SPC^3 PM Concurrent for 8 parallel threads

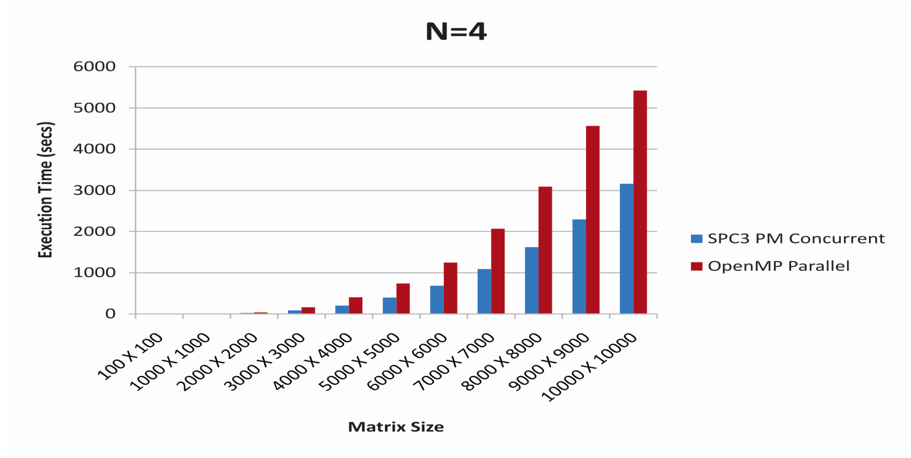
Matrix Size	Number of Parallel Threads	Execution Time (Sec)	
		OpenMP (Parallel)	SPC^3 PM, Concurrent
100 X 100	12	1	1
1000 X 1000		1	1
2000 X 2000		18	8
3000 X 3000		65	30
4000 X 4000		164	72
5000 X 5000		330	141
6000 X 6000		573	242
7000 X 7000		842	384
8000 X 8000		1291	575
9000 X 9000		1799	816
10000 X 10000		2664	1126

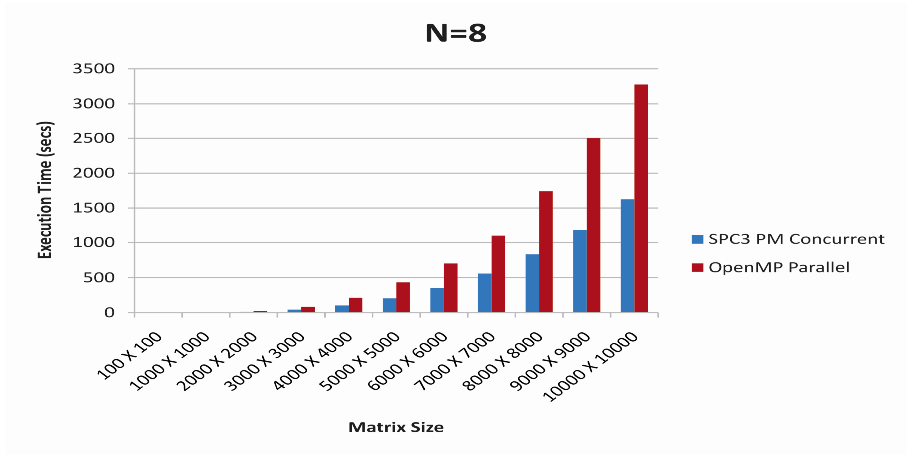
TABLE 4: Execution Time (Sec) for parallel matrix multiplication using OpenMP and SPC^3 PM Concurrent for 12 parallel threads

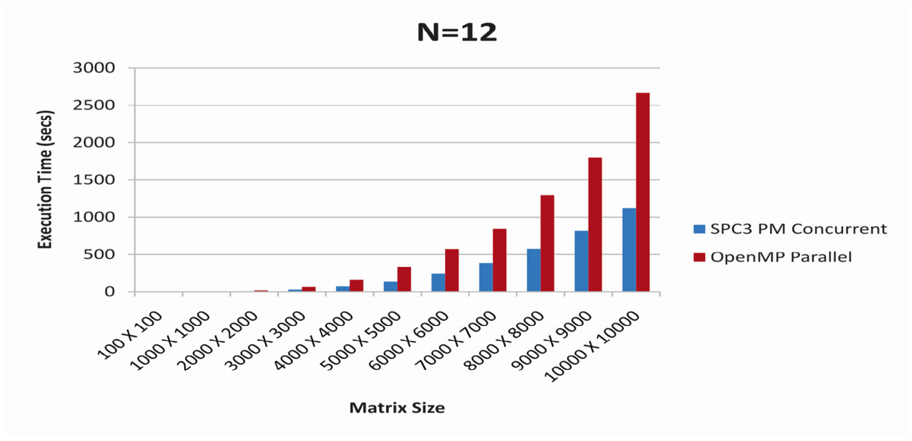
Matrix Size	Number of Parallel Threads	Execution Time (Sec)	
		OpenMP (Parallel)	SPC^3 PM, Concurrent
100 X 100	24	1	1
1000 X 1000		1	1
2000 X 2000		10	7
3000 X 3000		36	26
4000 X 4000		86	58
5000 X 5000		171	113
6000 X 6000		303	197
7000 X 7000		476	314
8000 X 8000		710	467
9000 X 9000		1011	661
10000 X 10000		1431	905

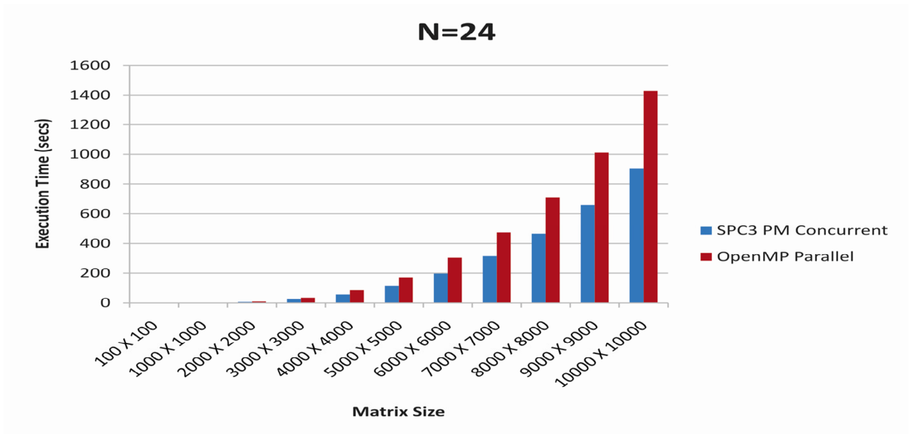
TABLE 5: Execution Time (Sec) for parallel matrix multiplication using OpenMP and SPC^3 PM Concurrent for 24 parallel threads

The figures 2-5 compare the execution time based on table 2-5 for each of the two approaches, OpenMP and SPC^3 PM Concurrent with different sizes of matrices for 4, 8, 12 and 24 parallel / concurrent threads respectively.









FIGURES 2-5: comparisons of the execution time for each of the two approaches, OpenMP and SPC^3 PM Concurrent

Based on table 1, the following table 6 shows the speedup obtained for the SPC^3 PM concurrent function for different matrices size and number of concurrent threads. Figure 6 shows the comparison of speedup based on table 2 for SPC^3 PM concurrent function with 4,8,12 and 24 concurrent threads.

Matrix Size	Time (Sec) Serial C++	Speedup			
		SPC^3 Concurrent N=4	SPC^3 Concurrent N=8	SPC^3 Concurrent N=12	SPC^3 Concurrent N=24
100 X 100	1	1.00	1.00	1.00	1.00
1000 X 1000	13	4.33	13.00	13.00	13.00
2000 X 2000	154	6.70	12.83	19.25	22.00
3000 X 3000	551	6.48	12.52	18.37	21.19
4000 X 4000	1374	6.80	13.21	19.08	23.69
5000 X 5000	2707	6.84	13.27	19.20	23.96
6000 X 6000	4642	6.81	13.23	19.18	23.56
7000 X 7000	7285	6.71	13.03	18.97	23.20
8000 X 8000	10925	6.75	13.12	19.00	23.39
9000 X 9000	15606	6.78	13.16	19.13	23.61
10000 X 10000	21497	6.80	13.22	19.09	23.75

TABLE 6: Speedup obtained for the SPC^3 PM concurrent function with different number of concurrent threads and matrices size

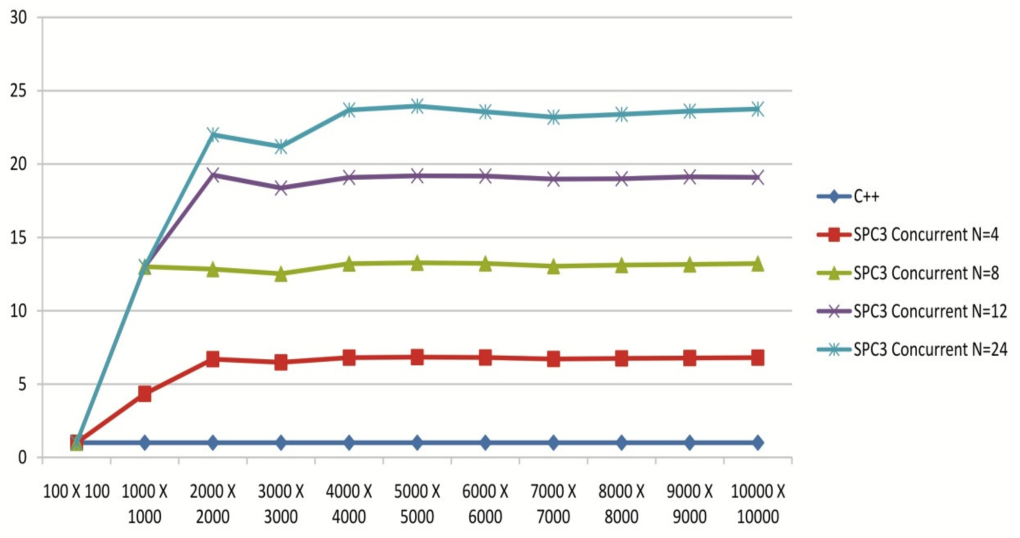


FIGURE 6: Comparison of speedup based on table 6 for SPC^3 PM concurrent function with 4,8,12 and 24 concurrent threads.

From the tables 2 to 5 and figures 2 to 5 it can clearly seen that concurrent function of SPC^3 PM takes much lesser execution time than that of the OpenMP. Another observation can be made that this performance gain increases as the either number of cores increases or the problem size increases. From Table 6 and figure 6 it can be clearly observed that the speedup obtained using SPC^3 PM are uniform and it is not affected by the problem size. These results proof the enhanced and scalable implementation of a concurrent matrix multiplication using SPC^3 PM.

The proposed concurrent matrix multiplication using SPC^3 PM provides an efficient, simple and scalable parallel matrix multiplication implementation for multi-core processors. The other parallel implementations of matrix multiplication proposed for SMPs are usually based on functional parallelism. The functional parallelism exploited alone with in a program cannot make most of multi-core processors. For maximum utilization of multi-core processors, program or programs in execution should be able to execute concurrently on available cores. The Concurrent function of SPC^3 PM provides the facility of concurrent execution of program or program on the available cores to maximize the utilization of multi-core processors. Besides, the existing parallel matrix multiplication algorithms for SMPs are very specific and difficult for common programmer to understand as they require detailed related subject knowledge whereas the proposed concurrent implementation of parallel matrix multiplication using SPC^3 PM is based on a simple and fundamental algorithm of matrix multiplication and does not requires any detailed related subject knowledge. In comparison to the attempts which have been made to solve matrix multiplication using data parallel or concurrent approaches on cell or GPUs, the proposed approach is more generic, architectural independent and found suitable for general purpose multi-core processors.

8. CONCLUSION AND FUTURE WORK

The results from this study show that the SPC^3 PM provides a simpler, effective and scalable way to perform matrix multiplication on multi-core processors. With the concurrent function of SPC^3 PM, the programmer can execute a simple and standard matrix multiplication algorithm concurrently on multi-core processors in much less time than that of standard parallel OpenMP. The proposed approach also shows more scalability, better and uniform speedup and better utilization of available cores than that of OpenMP. This SPC^3 PM will be further worked out for the introduction of some more parallel and concurrent functions and synchronizing tools.

9. REFERENCES

[1] N. Vachharajani, Y. Zhang and T. Jablin, "Revisiting the sequential programming model for the multicore era", *IEEE MICRO*, Jan - Feb 2008.

[2] M. D. McCool, "Scalable Programming Models for Massively Multicore Processors", *Proceedings of the IEEE*, vol. 96(5), 2008.

[3] G. Goumas, "Performance evaluation of the sparse matrix-vector multiplication on modern architectures", *Journal of Supercomputing*, pp. 1-42, Nov. 2008.

[4] R. Vuduc, H. Moon, "Fast sparse matrix-vector multiplication by exploiting variable block structure", *Lecture notes in computer science*, vol. 3726, pp. 807-816, 2005.

[5] H. T. Kung, C. E. Leiserson, "Algorithms for VLSI processor arrays"; in "Introduction to VLSI Systems", Addison-Wesley, 1979.

[6] G. C. Fox, S. W. Otto and A. J. G. Hey, "Matrix algorithms on a hypercube I: Matrix multiplication", *Parallel Computing*, vol. 4(1), pp.17-31, 1987.

[7] R. A. van de Geijn, J. Watts," SUMMA_ Scalable Universal Matrix Multiplication Algorithm", *TECHREPORT*, 1997.

[8] A. Ziad, M. Alqadi and M. M. El Emary, "Performance Analysis and Evaluation of Parallel Matrix Multiplication Algorithms", *World Applied Sciences Journal,* vol. 5 (2), pp. 211-214, 2008.

[9] Z. Alqadi and A. Abu-Jazzar, "Analysis of program methods used for optimizing matrix Multiplication*", Journal of Engineering*, vol. 15(1), pp. 73-78, 2005.

[10] J. Choi, "Fast Scalable Universal Matrix Multiplication Algorithm on Distributed-Memory Concurrent Computers" in Proceeding of *11th International Symposium on Parallel Processing IPPS '97 IEEE*, 1997.

[11] P. Alonso, R. Reddy, A. Lastovetsky, "Experimental Study of Six Different Implementations of Parallel Matrix Multiplication on Heterogeneous Computational Clusters of Multi-core Processors" in Proceedings of *Parallel, Distributed and Network-Based Processing (PDP)*, Pisa, Feb. 2010.

[12] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, P. Palkar, "A three-dimensional approach to parallel matrix multiplication", *IBM Journal of Research and Development*, vol. 39(5). pp. 575, 1995.

[13] A. Buluc, J. R. Gilbert, "Challenges and Advances in Parallel Sparse Matrix-Matrix Multiplication", in proceedings of *37th International Conference on Parallel Processing, ICPP '08,* Portland, Sep 2008.

[14] L . Buatois, Caumon, G. Lévy, "Concurrent number cruncher: An efficient sparse linear solver on the GPU", in Proceedings of the *High-Performance Computation Conference (HPCC)*, Springer LNCS, 2007.

[15] S. Sengupta, M. Harris, Y. Zhang, J.D. Owens, "Scan primitives for GPU computing". In *Proceedings of Graphics Hardware*", Aug. 2007.

[16] J. A. Stratton, S. S. Stone, Hwu, "M-CUDA: An efficient implementation of CUDA kernels on multicores", *IMPACT Technical Report 08-01*, University of Illinois at Urbana-Champaign, 2008.

[17] K. Fatahalian, J. Sugerman, P. Hanrahan, "Understanding the efficiency of GPU algorithms for matrix-matrix multiplication" in Proceeding of the *conference on Graphics hardware ACM SIGGRAPH/EUROGRAPHICS HWWS '04*, 2004.

[18] J. Bolz, I. Farmer, E. Grinspun, P. Schröoder, “Sparse matrix solvers on the GPU: conjugate gradients and multigrid”, *ACM Transactions on Graphics* (TOG), vol. 22(3), 2003.

[19] S. Ohshima, K. Kise, T. Katagiri and T. Yuba, “Parallel Processing of Matrix Multiplication in a CPU and GPU Heterogeneous Environment”, *High Performance Computing for Computational Science – VECPAR,* 2006.