MULTITHREADED FAST BLOCK (TILED) MATRIX MULTIPLICATION USING OPENMP

[Samina Haque]

[BTech] 3rd Year in Information Technology

Submitted as Project Thesis for Internship at

[Centre for Development of Advanced Computing, India]

[Institute Name – University of Calcutta]

[2020]

Keywords

1. Block Matrix
2. Matrix Multiplication
3. Memory Hierarchy
4. OpenMP
5. Parallel Programming
6. Tiled Matrix

Abstract

It is important to realize that performance optimizations can be very specific — they depend on the exact architecture of the machine (processor, memory, etc), the exact version of the compiler, the exact version of the operating system and the particular configuration of the program that we are trying to optimize. At present, Tiling is widely used by compilers and programmer to optimize scientific and engineering code for better performance. Many parallel programming languages support tile/tiling directly through first-class language constructs or library routines. However, the current OpenMP programming language is tile oblivious, although it is the de facto standard for writing parallel programs on shared memory systems. The increasing gap between memory latency and processor speed is a critical bottleneck in achieving high performance. To improve data access performance, one of the well-known optimization techniques is tiling. Tiling transforms loop nests so that temporal locality can be better exploited for a given cache size. However, tiling focuses only on the reduction of capacity cache misses by decreasing the working set size.

Table of Contents

Introduction

We use matrix multiplication as an example to illustrate tiled algorithms, a popular strategy to enhance locality of data access and enable effective use of shared memory. Block matrix multiplication enhances better locality through blocking. Tiled algorithms are an effective strategy for achieving high performance in virtually all types of parallel computing systems. The reason is that an application must exhibit locality in data access to make effective use of high-speed memories in these systems. For example, in a multicore CPU system, data locality allows an application to effectively use on-chip data caches to reduce memory access latency and achieve high performance. In parallel programming, tiling forces multiple threads to jointly focus on a subset of the input data at each phase of execution so that the subset data can be placed into these special memory types, consequently increasing the access speed.

1.1 BACKGROUND & CONTEXT

Parallel computing is about using many processors or multi-core CPU simultaneously to execute a program or multiple computational threads. Though, multi-core processors are widely available, but, the parallel programming is not that much popular among its users to harness the available multi-cores. Multi-core technology means more than one core inside a single chip. This allows multiple instructions of a program to be executed in parallel at the same time. Thread level parallelism (Multithreaded processors) executes multiple threads on multiple-cores in parallel and improves processor performance. A core is a part of the processor that performs read, execute, and write operations. One significant advantage of using Open MP is that the same source code can be used with Open MP compliant compilers and normal compilers as the Open MP commands and directives remain hidden to normal compilers.

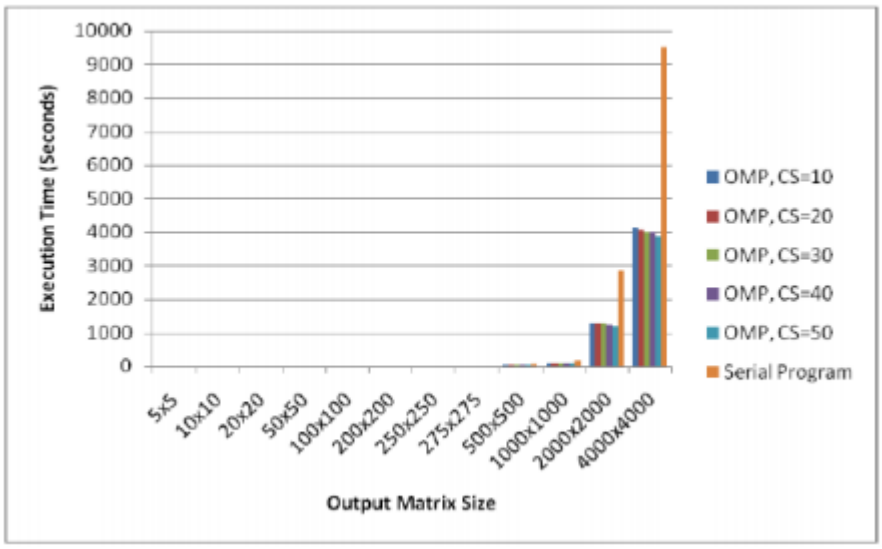


Figure: Execution time analysis of a parallel matrix multiplication program in OMP for different number of threads.

Table 5.2. Execution time analysis of a parallel matrix multiplication program in OMP for different number of threads.

Output matrix size	Execution time (seconds) with OMP			
	nt=4	nt=10	nt=25	nt=50
5x5	0.079	0.078	0.078	0.078
10x10	0.08	0.08	0.082	0.081
20x20	0.125	0.124	0.125	0.125
50x50	0.461	0.454	0.451	0.449
100x100	1.258	1.121	1.078	0.992
200x200	3.976	3.276	3.082	2.882
250x250	6.4	6.012	5.8	5.376
275x275	7.45	7.163	6.763	6.452
500x500	24.26	22.539	20.373	17.34
1000x1000	80.76	76.647	71.243	64.39
2000x2000	1208.56	1178.549	1109.564	1012.28
4000x4000	3878.47	3767.645	3665.352	3441.43

To achieve the necessary reuse of data in local memory, researchers have developed many new methods for computation involving matrices and other data arrays. Typically, an algorithm that refers to individual elements is replaced by one that operates on subarrays of data, which are called *blocks* in the matrix computing field. The operations on subarrays can be expressed in the usual way. The advantage of this approach is that the small blocks can be moved into the fast-local memory and their elements can then be repeatedly used.

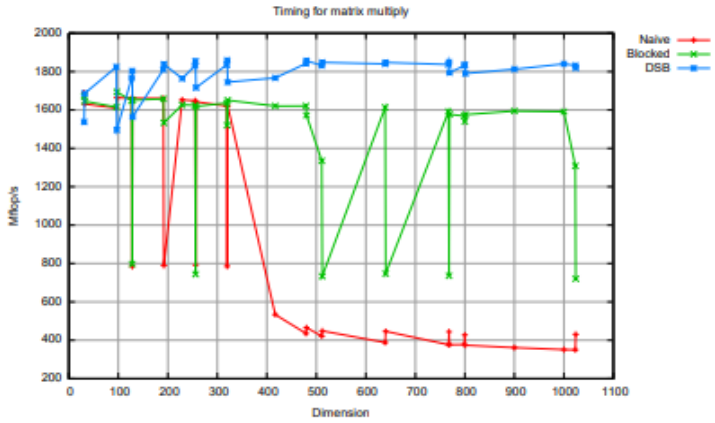
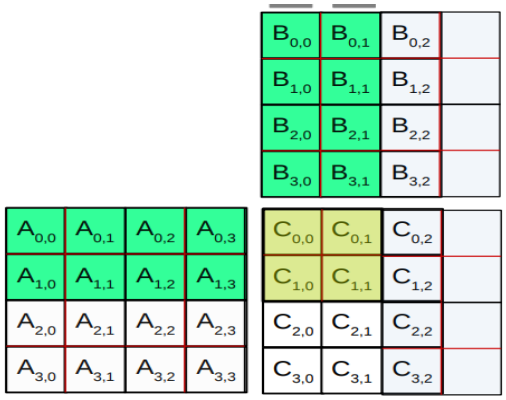


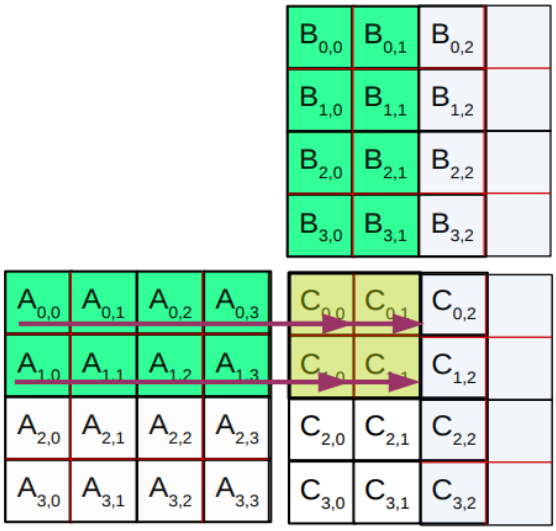
Fig: Better Locality is achieved through blocking

Shared Memory Tiling / Blocking

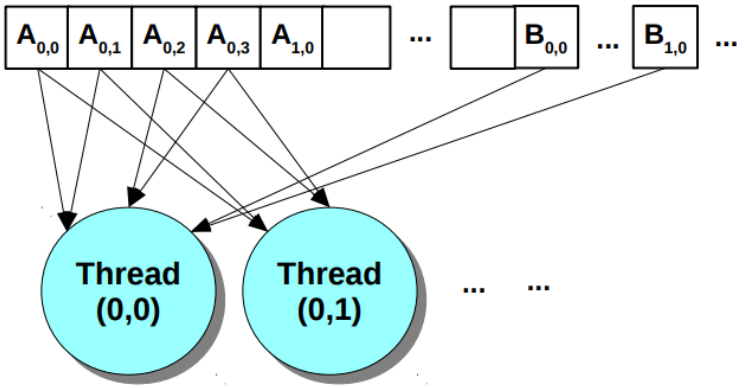
Considering, two matrices A and B and their result is C,



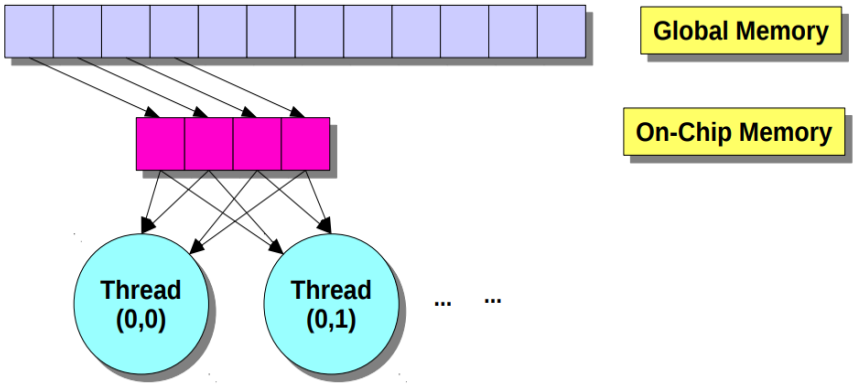
Work in block(0,0):



Now, considering threads 0 and 1;



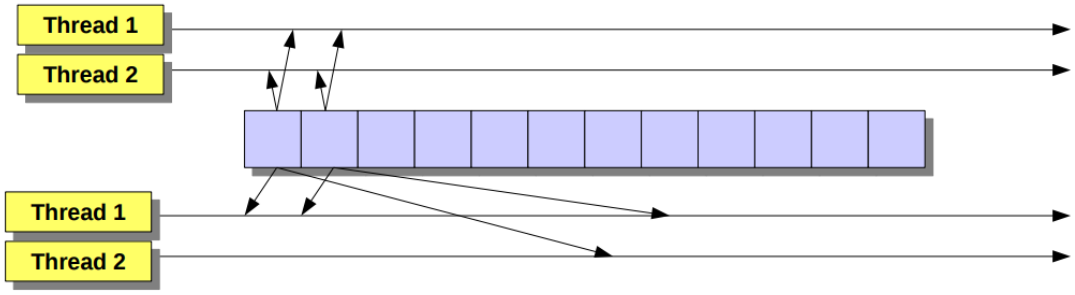
All threads can access global memory to input matrix elements.



Here, we divided the global memory content into tiles and focusing on the computation of one or small number of tiles at each point of time.

Some basic concepts of tiling:

- if tiled data exhibit good spatial locality then it is more efficient.
- Secondary Memory should be large enough to accommodate all the data elements.
- Synchronization is needed.



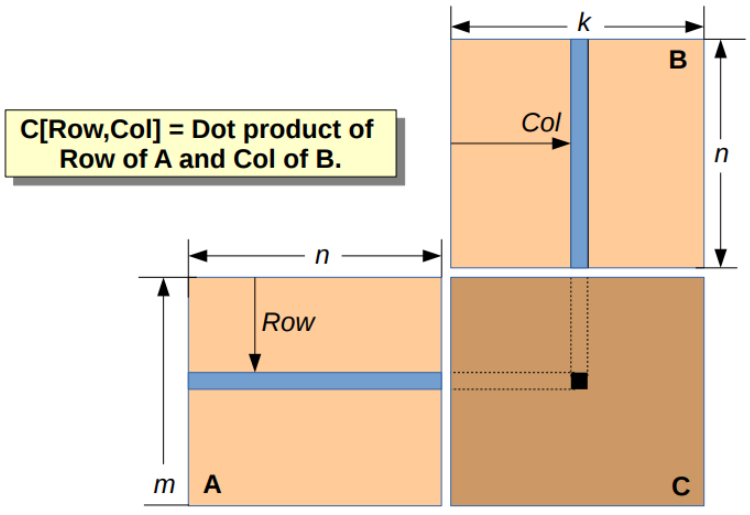
Tiling Technique

- Identify a tile of global memory content that are accessed by multiple threads.
- Load the tile from global memory into on-chip memory
- Have the multiple threads to access their data from on-chip memory.
- Then shift to the next tile.

Tiled Matrix Multiplication

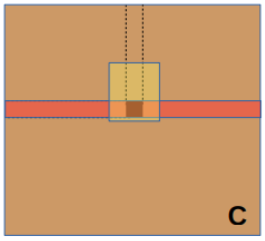
- Loading a tile
- Phased Execution
- Barrier Synchronization

Basic Matrix Multiplication:

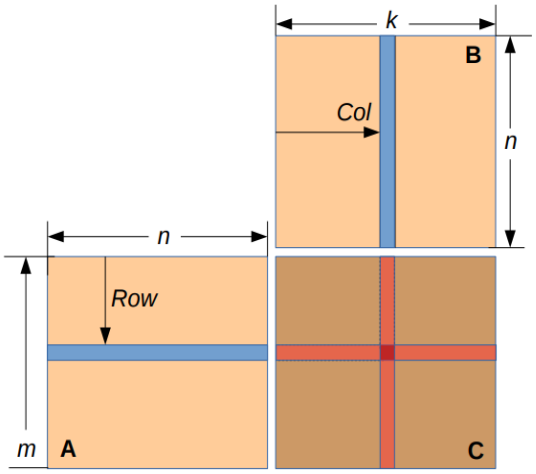


- Elements A[Row,…] are used in calculating elements C[Row,…]

- All threads updating C[Row,…] will access A[Row,…]

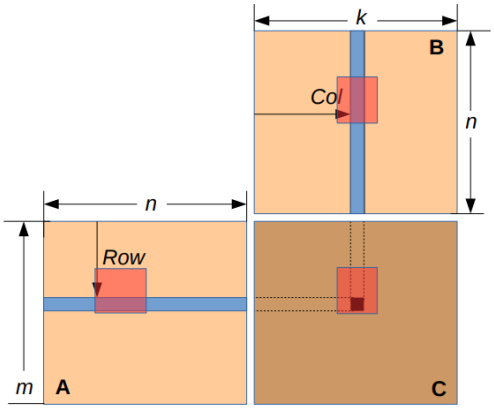


- Each thread in the block loads A[Row]



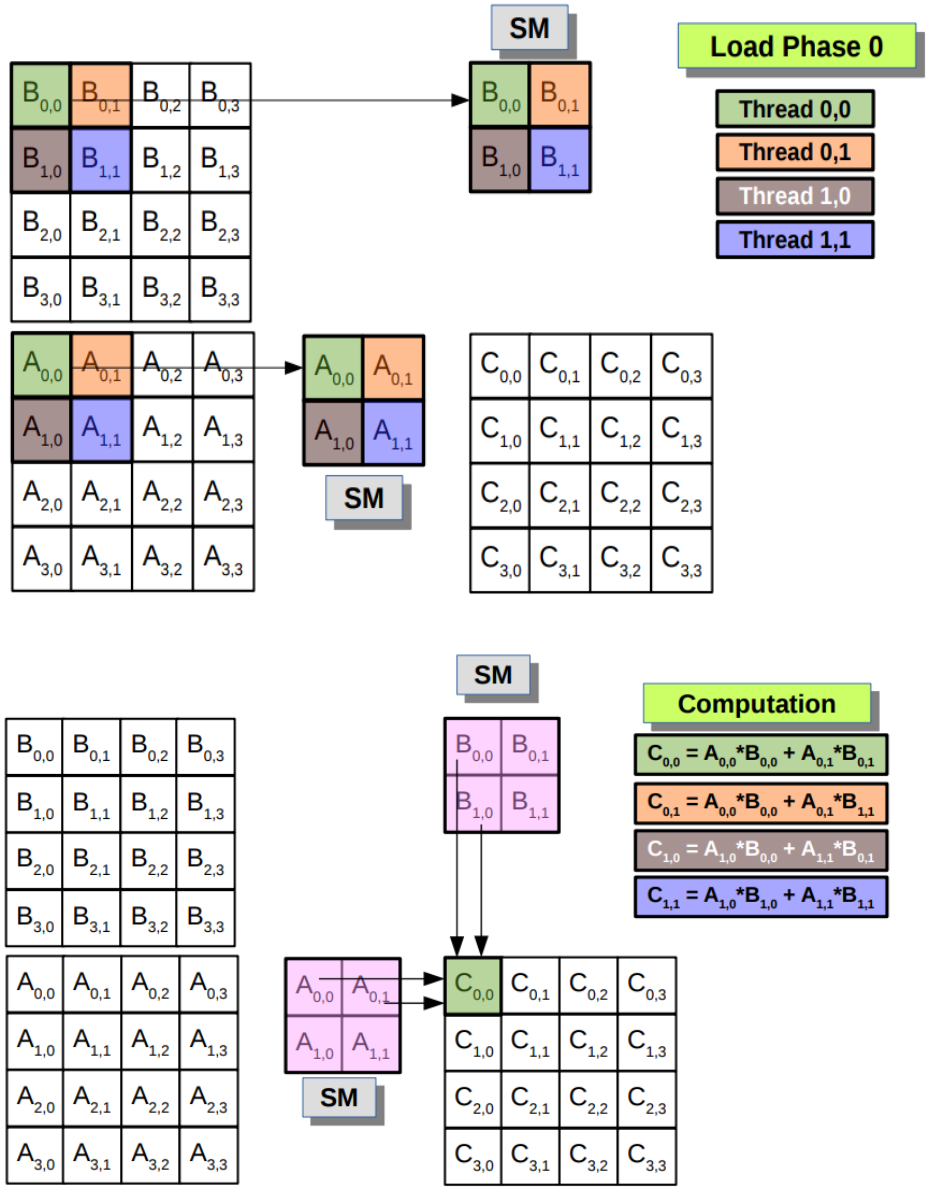
- Elements B[...,Col] are used in calculating elements C[...,Col]
- Memory Bandwidth reduction is by 50%

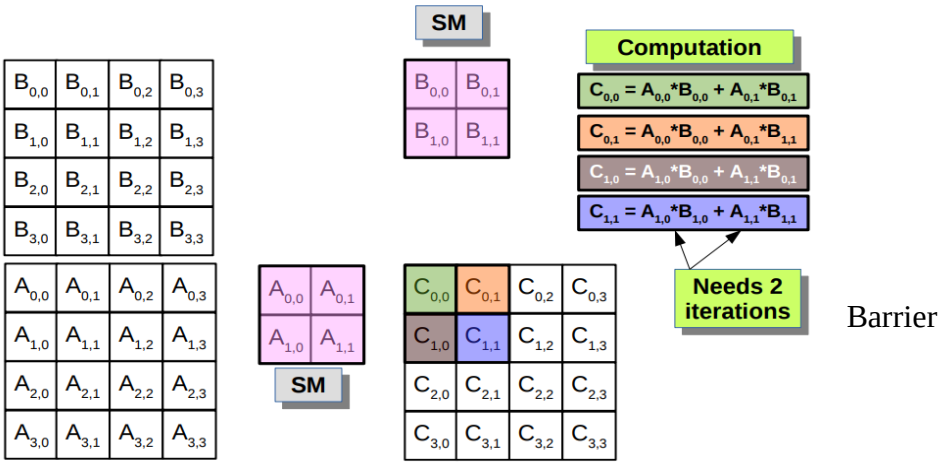
Tiled Matrix Multiplication



All threads in a block participate

– Each thread loads one A element and one B element in tiled code





Synchronization

- API call: __syncthreads()
- All threads in the same block must reach the __syncthreads() before any can move on
- Best used to coordinate tiled algorithms
  - To ensure that all elements of a tile are loaded
  - To ensure that all elements of a tile are consumed
- Barriers can significantly reduce active threads in a block

1.2 PURPOSES

Tiling has been used as an effective compiler optimizing technique to generate high performance scientific codes. Tiling not only can improve data locality for both the sequential and parallel programs, but also can help the compiler to maximize parallelism and minimize synchronization for programs running on parallel machines. Tiling is essentially a program design paradigm. It is a natural representation for many important data objects that are heavily used in scientific and engineering algorithms. Scientific code that is written with the concept of tile/tiling in mind usually looks concise and clear, and thus is much easier to understand and less error prone.

1.3 METHODOLOGY AND RESEARCH DESIGN

Methodology

To compile and execute the code, OpenMP installation was done on Ubuntu platform. After creating the script in vim editor, compilation is done with -fopen construct. After successful compilation, execution is done ./a.out

Code

```
#include <stdio.h>
#include<stdlib.h>
#include <omp.h>
#include <math.h>
#include <time.h>

#pragma comment (lib, "msmpi.lib")

int main(int argc, char **argv)
{
	int size, block_size;
	printf("\nEnter the size of matrix\n");
	scanf("%d",&size);
	printf("\nEnter the size of block\n");
	scanf("%d",&block_size);
	int A[size][size],B[size][size],C[size][size],CC[size][size],BT[size][size];

	for (int i = 0; i < size; i++)
	{
	 for (int j = 0; j < size; j++)
	 {
		A[i][j] = (rand()%5);
		B[i][j] = (rand()%5);
		C[i][j] = 0;
		CC[i][j] = 0;
	 }
	}

	for (int i = 0; i < size; i++)
	{
	 for (int j = 0; j < size; j++)
	 {
		BT[i][j] = B[j][i];
	 }
	}

	int i = 0, j = 0, k = 0, jj = 0, kk = 0;

	//serial
	clock_t t1,t2;
	printf("\n\nComputation has begun(Serial block)\n");
	t1= clock();
```

```
            for (jj = 0; jj < size; jj += block_size)
            {
              for (kk = 0; kk < size; kk += block_size)
              {
                        for (i = 0; i < size; i++)
                        {
                         for (j = jj; j < ((jj + block_size) > size ? size : (jj + block_size)); j++)
                        {
                                int tmp = 0;

                        for (k = kk; k < ((kk + block_size) > size ? size : (kk + block_size)); k++)
                        {
                                tmp += A[i][k] * B[k][j];
                        }
                                C[i][j] += tmp;
                        }
            }
        }
}
t2= clock();
printf("time taken by serial computing %f\n",((float)(t2-t1)/CLOCKS_PER_SEC));

for ( i = 0; i < size; i++)
{
            for ( j = 0; j < size; j++)
            {
                        C[i][j] = 0;
                        CC[i][j] = 0;
            }
}

//parallel block
printf("\n\nComputation has begun(parallel block)\n");
t1= clock();
int tmp;
int chunk = 1;
int tid;
omp_set_num_threads(4);

#pragma omp parallel shared(A, B, C, size, chunk) private(i, j, k, jj, kk, tid, tmp)
{
            //omp_set_dynamic(0);

            #pragma omp for schedule (static, chunk)
            for (jj = 0; jj < size; jj += block_size)
            {
               for (kk = 0; kk < size; kk += block_size)
              {
                  for (i = 0; i < size; i++)
                  {
                        for (j = jj; j < ((jj + block_size) > size ? size : (jj + block_size)); j++)
                        {
                            tmp = 0;
                        for (k = kk; k < ((kk + block_size) > size ? size : (kk + block_size)); k++)
                        {
                                tmp += A[i][k] * B[k][j];
                        }
```

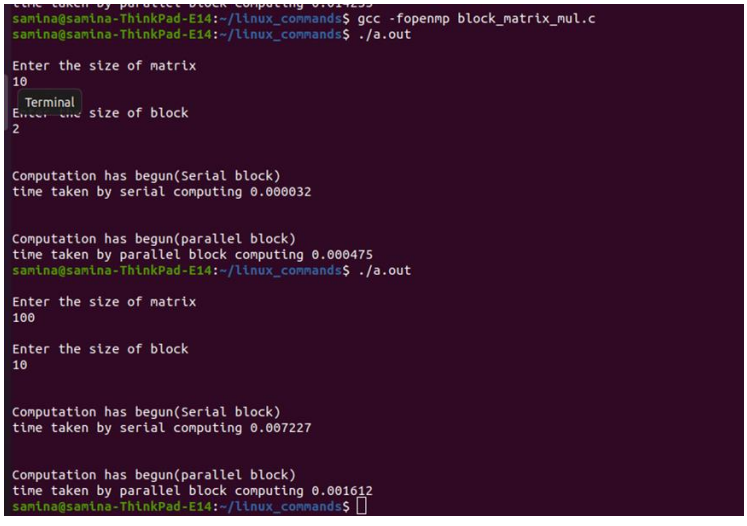
```
                        C[i][j] += tmp;
                        }
                }
        }
  }
}
t2= clock();
printf("time taken by parallel block computing %f\n",((float)(t2-t1)/CLOCKS_PER_SEC));

return 0;
}
```

1.4 RESULTS

Size of matrix given by user – 100

Size of block given by user - 10



1.5 ANALYSIS

The code very explicitly shows that the execution time differs clearly while in computation in serial and parallel block. Much less time is required in parallel block computation which is what was the desired output.

Conclusion

Tiling has proven to be an effective mechanism to develop high performance implementations of algorithms. Tiling can be used to organize computations so that communication costs in parallel programs are reduced and locality in sequential codes or sequential components of parallel programs is enhanced. This project exactly focuses on this how the execution time is essentially dropped down. With the increase in the size of matrix, the difference in computation time in case of serial and parallel block computing is prominent enough, that is, there is effective decrease in time while computing in parallel block. Also, this project can create a framework for application developers to exploit features of advanced architectures to achieve high performance.

References

1. Tiling, Block Data Layout, and Memory Hierarchy Performance Neungsoo Park, Member, IEEE, Bo Hong, and Viktor K. Prasanna, Fellow, IEEE
2. Tuning matrix multiplication –

   James Demmel http://www.cs.berkeley.edu/~demmel/cs267_Spr16/
3. Patterns for Cache Optimizations on Multi-Processor Machines Nicholas Chen, Ralph Johnson Department of Computer Science University of Illinois at Urbana-Champaign {nchen, johnson}@cs.illinois.edu
4. Programming for Parallelism and Locality with Hierarchically Tiled Arrays Ganesh Bikshandi, Jia Guo, Daniel Hoeflinger, Gheorghe Almasi† , Basilio B. Fraguela‡ , Mar´ıa J. Garzar´an, David Padua and Christoph von Praun† University of Illinois at Urbana-Champaign bikshand, jiaguo, hoefling, garzaran, padua@cs.uiuc.edu † IBM T.J. Watson Research Center gheorghe, praun@us.ibm.com ‡Universidade da Coru˜na,Spain basilio@udc.es
5. https://aranisumedh.wordpress.com/2017/04/20/part-3-parallel-tile-reduction/#:~:text=This%20is%20the%20third%20and,programs%20on%20shared%20memory%20systems.
6. A-comparison-of-blocked-and-cyclic-tiling-techniques-for-multiple-threads-The-blocked_fig3_2765526