LAB 2: STUDYING THE CACHE IMPACT OF MULTICORE PROCESSORS

Goal

In this lab, you will compile and run the different versions of Matrix Multiplication programs on the PDS Lab's cluster to study about the **cache effects on sequential and parallel programs running on multi-core processors**. You also learn the skill of profiling a program for performance analysis.

Submission

- Submit your lab answers as Word documents (.doc, .docx, .xls) attachments to Canvas. You can answer the lab questions directly in this file or in a separate file. Make sure the submitted file's name is "Lab2_firstname_lastname.docx".

Part 1. Understands different versions of Matrix Multiplication.

The Matrix Multiplication was implemented in four different versions. Each version is a function in the source code file **mm1.cpp**.

$$\mathbf{C= A * B =} \begin{vmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{vmatrix} * \begin{vmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \dots & \dots & \dots & \dots \\ b_{n1} & b_{n2} & \dots & b_{nn} \end{vmatrix} = \begin{vmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \dots & \dots & \dots & \dots \\ c_{n1} & c_{n2} & \dots & c_{nn} \end{vmatrix}$$

- **Version1**: The function ***SequentialMatrixMultiplication_Version1*** in source code file

```
Pseudo Code
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            $c_{ij} = c_{ij} + a_{ik}*b_{kj}$
```

- **Version 2**: The function ***SequentialMatrixMultiplication_Version2*** in source code file

```
Pseudo Code
for i = 1 to n
    for k = 1 to n
        for j = 1 to n
            $c_{ij} = c_{ij} + a_{ik}*b_{kj}$
```

Version 3: The function ***ParallelMatrixMultiplication_Version3*** in source code file

- This function is a parallel version of the version 1. The **#pragma omp parallel for** directive indicates that the following loop is executed in parallel.

Version 4: The function ***ParallelMatrixMultiplication_Version3*** in source code file

- This function is a parallel version of the version 2. The **#pragma omp parallel for** directive indicates that the following loop is executed in parallel.

Question 1 (1pt). What is the main difference between version 1 and version 2? Which version better leverages caches and why? (you don't need to run the program to answer this question at this point. However, you may need to look into the given codes to understand the differences between the two versions).

The main difference is in the order that version 1 and version 2 run their inner loop. This in turn affects how the matrix multiplication is applied. Version 1 runs [i]**[j]** += a[i][k]*b[k]**[j]**; and version 2 runs [i][j] += a[i]**[k]***b**[k]**[j]; There is a loop interchange between the innermost loops.

I bolded where the 2nd loop runs and so because version 1 has loop J running 2nd. The swapping of the 2nd line affects the multiplication because A[i][k] is in cache but B[k][j] are not, so there's a miss penalty on B.

On version 2, the speedup then becomes noticeable. Both A and B are in cache so there is less miss penalty.

Spatial locality is improved in version 2. Every time a block of memory is loaded, the misses are reduced because of the interchange.

Question 2 (1pt). How many threads the parallel versions (version 3 and 4) are using ? Explain it in terms of physical core, logical core and hyper threads.
Versions 3 and 4 have 48 logical cores, 24 physical threads and Open MP with 48 Threads. Hyper threading is when each core can run two threads simultaneously. It converts a single physical processor into 2 virtual processors that gives 48 threads. Per node, it can run 2 processors, 12 cores per processor and 2 threads per core. 12 cores per processor and 2 threads per core would equal 24 physical threads.

Part 2: Analysis

1. (4 pt) For each version of matrix multiplication, run the program with matrix sizes according to Table 1. For each matrix size, you should run the program at least 2 times and average the results. Then fill in Table 1 as well as plot the runtime for Heracles in a chart (x-axis for the matrix size, y-axis for the runtime)

   Compiling C/C++ program: Read Section 1 for how to compile a C/C++ program on Heracles. You can also read the instructions from http://pds.ucdenver.edu/webclass/index.html/

   Note: The Matrix Multiplication program in this lab gets three arguments as its inputs:

   - 1st argument: the size of the matrix
   - 2nd argument: the version number of the program, 1 = version 1, 2 = version 2, etc.
   - 3rd argument: the option to specify if input/output matrices are printed or not (1=print; 0= not print). Program only prints if the matrix size is less than 10.

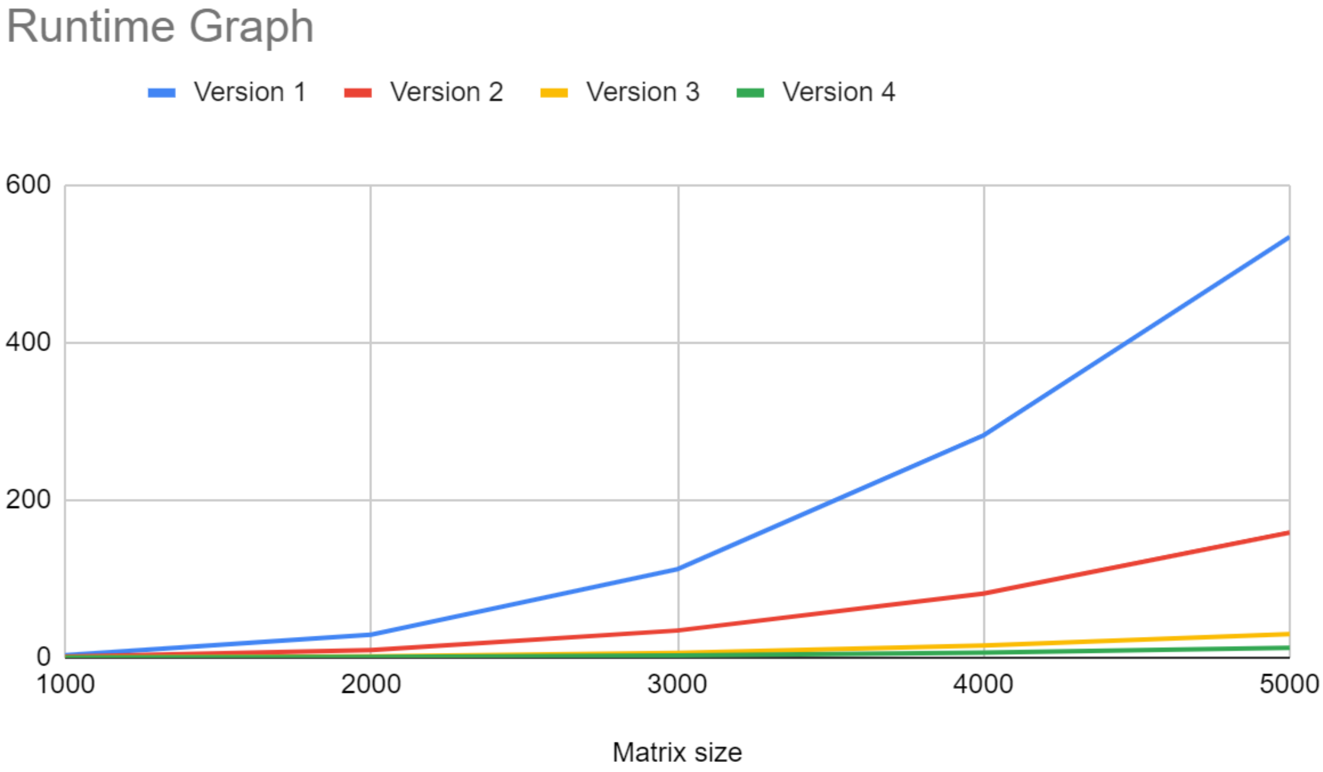
Table 1 – Heracles results

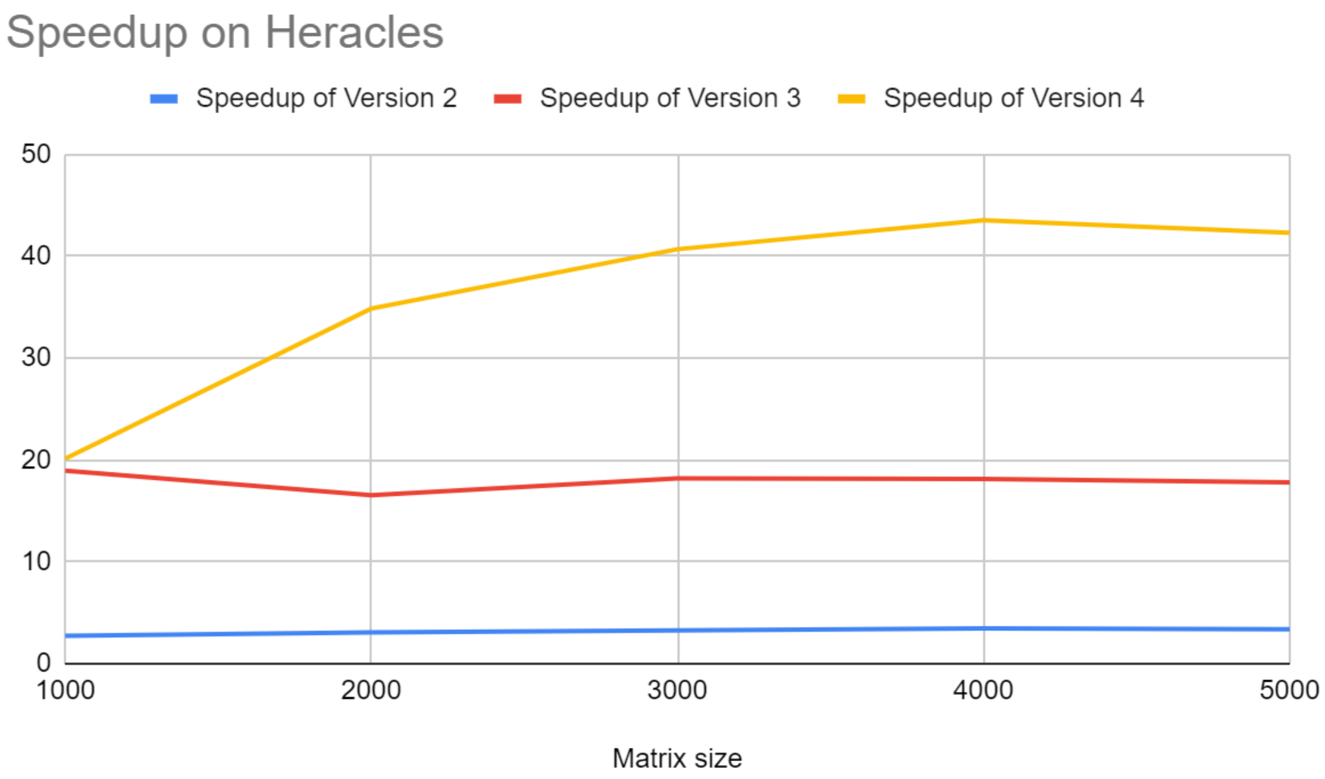
Matrix size	Runtime of Version 1	Runtime of Version 2	Runtime of Version 3	Runtime of Version 4
1000	3.22	1.18	0.17	0.16
2000	29.29	9.58	1.77	0.84
3000	112.32	34.42	6.17	2.76
4000	282.18	81.39	15.57	6.48
5000	533.97	158.70	30.00	12.62

Table 2.1 - Speedup on Heracles

Matrix size	Speedup of Version 2	Speedup of Version 3	Speedup of Version 4
1000	2.728813559	18.94117647	20.125
2000	3.057411273	16.5480226	34.86904762
3000	3.263219059	18.20421394	40.69565217
4000	3.467010689	18.12331407	43.5462963
5000	3.364650284	17.799	42.31141046

Plot the runtime graph here for Heracles (Table 2) in a chart (x-axis for the matrix size, y-axis for the runtime)
So this is confusing. Do we plot the runtime graph? Which is Table 1 or do we plot Table 2, which is speedup? I emailed Thoria and got no response so here's both for lack of clarification





2. **Observe cache misses of different matrix multiplication on Heracles.** In order to profile to observe cache misses of a program, you will use **Perf** profiler. **Perf** profiler can be used to measure not only cache misses but also other information such as CPU clocks, branch prediction, etc.
   More information of **Perf** can be found at http://www.brendangregg.com/perf.html, https://perf.wiki.kernel.org/index.php/Main_Page and, http://www.pixelbeat.org/programming/profiling/

   The perf command of the Profiling C/C++ programs on Heracles is as follows:

   # ssh node<#n> perf stat -e <event1> -e <event2> -e <event n> <path/>Myprogram arg1 arg2 argn

   - Perf: profile tool that will monitor your program.
   - event: predefined events displayed in perf list command.
   - #n: is the node number (0, 1, 2, 3, ..., n) that you'd like to run your program on.

     Example:

     ssh node2 perf stat -e cache-misses /myPath/mm 4 2 1

   For this question, you have to profile only the cache misses for all the versions and sizes. Fill in your results in Table 3.

Table 2 – Profile cache misses on Heracles.

Matrix Size	Version 1	Version 2	Version 3	Version 4
	Cache misses	Cache misses	Cache misses	Cache misses
1000	86,396	81,855	453,411	471,962
2000	330,444,292	222,758,757	20,307,208	23,721,035
3000	3,379,369,702	3,166,918,172	167,620,855	229,925,013
4000	8,015,828,002	7,523,154,145	376,210,217	845,622,086
5000	15.644.324.445	14,697,677,586	695,699,224	2,412,843,386

Answer the following questions based in the experiments you've done in this part.

Question 3 (1pt). What is the main factor for the runtime difference between Version 1 and Version 3? Why?

The main factor for runtime difference between version 1 and version 3 is parallelism. Version 3 is running the **#pragma omp parallel** indicates that the loop is executed in parallel. Version 1 is running sequentially. So because Version 3 is running in parallel, this shortens up the time since multiple things can be running at once.

Question 4 (1pt). What is the ideal speedup when you run a program that uses N cores? Does version 4 achieve the ideal speedup? Provide results based on your experiments.

The ideal speedup when you run a program should be N times. The speedup of version 4 is at least 40+x times faster than version 1 at a matrix size of 5000. On top of version 4 running in parallel, which also activates hyper threading as the matrix size goes up from 1000 to 5000. Version 4 uses 48 threads in parallel which would be close to about 40x times faster which was what was calculated in the speedup table.

Question 5 (1pt). Compare the speedups of version 2, 3, and 4 and explain why these versions have different speedup? (Hint: Which techniques are used in version 2, 3, and 4?). Provide experiment results to support your answers.

Version 2: Uses sequential execution, as it was explained in an earlier question, the speedup then becomes noticeable. Both A and B are in cache so there is less miss penalty. The threads take advantage of spatial locality. However, because it is sequential, it is slower than versions 3 and 4.

Version 3: Uses parallel execution, the loops of j and k are normal, allowing for more cache misses. The loops run sequentially so this allows for more misses.

Version 4: Uses parallel execution, the loops of j and k are switched, reducing cache misses. The elements are in cache consistently and are more easily accessible, making the speedup time a lot faster. The spatial locality is improved because of the loop interchange between versions 3 and 4

Since version 3 and 4 are running in parallel, it makes full use of hyper threading, allowing for faster run time

Question 6 (1pt). When the matrix size increases, what are the factors resulting in the speed up of the Version 4?

Parallelism results in the speedup of version 4 as matrix size increases. Also the fact that there are less cache misses also contributes to speed up and shorter runtime. Spatial locality optimization is also a factor since there are more cache hits, the process can complete and return the physical address to memory rather than having to go through a cache miss or even page fault where it needs to go to secondary storage to find the proper pages.

Section 1. COMPILING AND RUNNING THE PROGRAMS ON HERACLES

- Logon the Heracles server (see Lab 1)
- Make a folder named csc5593/lab3 in your home directory using the **mkdir** command.
- Copy the source code files to csc5551 or csc7551. If you are using MAC or Linux, you can copy these files using **scp** or **sftp** command. If you are using Windows, you can use WinSCP or SSH Secure Shell to login and copy files from your PC to the cluster. For more information on downloading WinSCP and how to use this software, visit:

http://pds.ucdenver.edu/document.php?type=software&name=winscp

- Hardware Information about Heracles:

http://pds.ucdenver.edu/webclass/Heracles_Architecture.html

- Compiling C++/openMP programs on Heracles

http://pds.ucdenver.edu/webclass/Compiling%20openMP%20programs.html

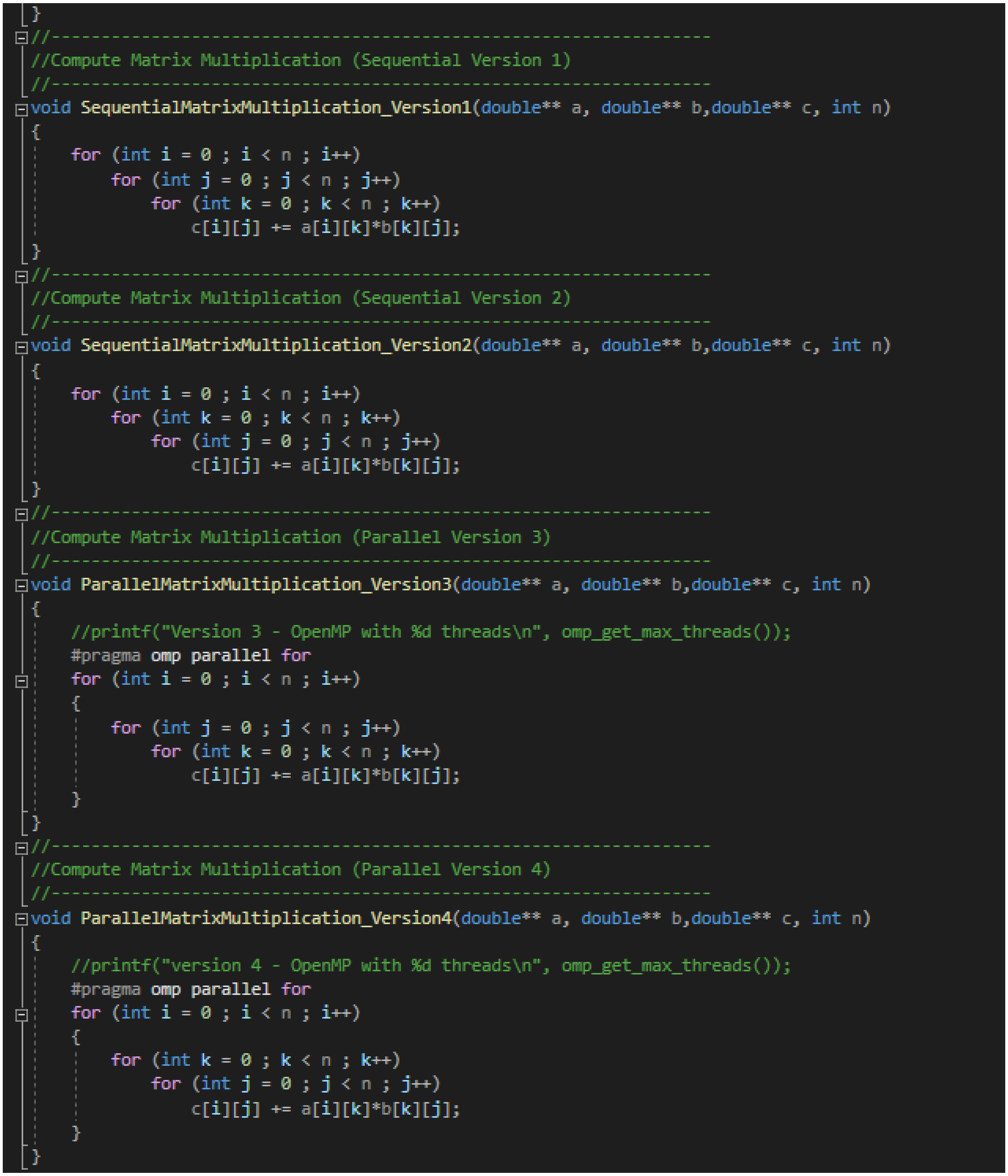
- Running program on Heracles

  The first step when running your program on a compute node on Heracles is to find an open node to run on. Visit https://heracles.ucdenver.pvt/mcms/ for monitoring the compute nodes on Heracles. Try to select a node that shows the lowest usage.

  http://pds.ucdenver.edu/webclass/Heracles-RunningPrograms.html

  Scheduling jobs on Heracles by using SLURM

  http://pds.ucdenver.edu/webclass/Heracles-RunningPrograms%20Slurm.html



```
}
//------------------------------------------------------------------
//Compute Matrix Multiplication (Sequential Version 1)
//------------------------------------------------------------------
void SequentialMatrixMultiplication_Version1(double** a, double** b,double** c, int n)
{
    for (int i = 0 ; i < n ; i++)
        for (int j = 0 ; j < n ; j++)
            for (int k = 0 ; k < n ; k++)
                c[i][j] += a[i][k]*b[k][j];
}
//------------------------------------------------------------------
//Compute Matrix Multiplication (Sequential Version 2)
//------------------------------------------------------------------
void SequentialMatrixMultiplication_Version2(double** a, double** b,double** c, int n)
{
    for (int i = 0 ; i < n ; i++)
        for (int k = 0 ; k < n ; k++)
            for (int j = 0 ; j < n ; j++)
                c[i][j] += a[i][k]*b[k][j];
}
//------------------------------------------------------------------
//Compute Matrix Multiplication (Parallel Version 3)
//------------------------------------------------------------------
void ParallelMatrixMultiplication_Version3(double** a, double** b,double** c, int n)
{
    //printf("Version 3 - OpenMP with %d threads\n", omp_get_max_threads());
    #pragma omp parallel for
    for (int i = 0 ; i < n ; i++)
    {
        for (int j = 0 ; j < n ; j++)
            for (int k = 0 ; k < n ; k++)
                c[i][j] += a[i][k]*b[k][j];
    }
}
//------------------------------------------------------------------
//Compute Matrix Multiplication (Parallel Version 4)
//------------------------------------------------------------------
void ParallelMatrixMultiplication_Version4(double** a, double** b,double** c, int n)
{
    //printf("version 4 - OpenMP with %d threads\n", omp_get_max_threads());
    #pragma omp parallel for
    for (int i = 0 ; i < n ; i++)
    {
        for (int k = 0 ; k < n ; k++)
            for (int j = 0 ; j < n ; j++)
                c[i][j] += a[i][k]*b[k][j];
    }
}
```