

CA670 Concurrent Programming

Name	Aditya Gupta
Student no.	19210195
Programme	MSc. Cloud Computing
Module Code	CA670
Assignment title	Assignment 2- OpenMP
Submission Date	17 th April, 2020
Module Coordinator	David Sinclair

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. I have read and understood the Assignment Regulations set out in the module documentation. I have identified and included the source of all facts, ideas, opinions, and viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged, and the source cited are identified in the assignment references. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

I have read and understood the referencing guidelines found recommended in the assignment guidelines.

Name: Aditya Gupta

Date: 17 April 2020

Efficient Large Matrix Multiplication in OpenMP

Introduction

In this assignment, I have developed an efficient large matrix multiplication algorithm in OpenMP. So, what is OpenMP? It is an Application Programming Interface for shared memory model programming. It consists of compiler directives, and that is the `#pragma omp parallel`. These are the directives for the compiler. These are picked up by the compiler when we compile the code and replaced with some other code, so the compiler figures out what to put over there. Then, there are runtime library routines these represent functions that can be invoked within your code. Consequently, there is a library that is linked along with your code the openMP library, and it provides support for all these functions so examples for this are `'omp_get_num_threads()'`, `'omp_get_thread_num()'` and so on. Therefore, these are functions, and these are also part of openMP. Finally, we have environment variables, and the example of this is `'omp_num_threads'` and that sets the number of threads that you want to execute in parallel region.

So, what are the advantages of OpenMP? Well, standardization and portability remain the most important ones. Hence, this means that we can carry our code from one platform to another platform and there is no need to alter the code again. When you move from the GCC compiler to any compiler, you do not have to change the code again. OpenMP is a standard which if a compiler says, 'it supports', it supports the entire set. There are different versions of OpenMP, so we need to be aware of which version is being supported and what are the features being provided by that version and no matter which compiler you use which supports that version of OpenMP (it) the code will compile and run properly so that is the most considerable advantage, and of course, ease of use.

Therefore, OpenMP is very simple in the sense that we can normally start with the code that we had, the sequential code that we had and we can just compile it with OpenMP and run - we just have to "include", it will run, just that it will run sequentially and now if we want to parallelize it then just by figuring out how to parallelize that part; and don't have to touch the remaining code this is a huge advantage. And that makes parallelizing codes very simple. You can start with sequential code. If you go to something like MPI - message passing interface, you must rewrite the entire code before you can run it in parallel, that is a challenge with message-passing programs. Hence, that is the advantage of shared memory programming and OpenMP, of course. We can incrementally parallelize the code start with a sequential code and then incrementally pick up one part then another part and so on and parallelize it.

OpenMP has been around since 1997, and that was (version 1.0), the latest one is (version 4). It is a thread-based model so there are threads that share the same address space. Accordingly, everything is shared by the threads, except for their stacks and the thread-local space. So, all the global variables, the code everything is shared. Ultimately, OpenMP is based on the Fork-Join model. So what is the Fork-Join model - when you encounter the `'#pragma omp parallel'` region, when you encountered this compiler directive, at that time what happens is, the compiler replaces

this with the code that creates threads so up to this point, up to omp parallel point, code is executing sequentially, there is a separate thread that is executing. So, let us say, 'omp_num_threads' is set to 4. As a result, what will it perform? It will actually — at this point in time — it will launch three threads. It will launch three new threads, and one of the threads is going to be the thread that was already executing. So, this thread, which was already executing, continues as one thread and three additional threads get launched at this point, and then the compiler sets them up so that they execute this code which is there in the parallel region. So, all of them execute this parallel region and when we encounter this parenthesis close, when the parallel region ends, at that point of time a join operation is performed. This is called forking, and, in the end, a join operation is performed, where essentially the master thread it will, at this point, wait for all the other threads to complete. So, all the other threads, when they complete, they will come and join. And when all the other threads have completed, they have joined back, at that time it will proceed ahead again with the remaining code. Hence, this is called as the fork-join model.

The Code

In this we are performing blocked matrix multiplication algorithm as blocking is a well-known optimization technique for improving the effectiveness of memory hierarchies. There is a slight variation from the normal matrix multiplication in this we divide the matrix into smaller sub matrices and then calculate those matrices individually. The main advantage of this algorithm is that it will store the values that we need to do matrix multiplication in cache memory so this will speed up the process. So, instead of multiplying entire rows or columns of an array the block multiplication algorithm operates on the specific blocks or sub matrices. I did the implementation of large matrix multiplication using block algorithm in OpenMP using C language. So, we have 4 files here attached in this zip folder named as 'CA670' :

1. **Matrix.h:** In this file, we have header files including all the libraries and required packages so in starting of the code I have included

```
#ifndef HEADER
#define HEADER
#endif
```

This is just a macro used for conditional compilation, this is effective when having multiple header files where we can include header more than twice. And in between this macro, we have included our header files including 'omp.h' which comprise a library for parallel programming and used in OpenMP and defined Minimum Dimension (MIN_D) for the matrix as 960 and Maximum Dimension (MAX_D) for the matrix as 1472 so it will give the output for the range from 960 * 960 to 1472 * 1472 in the steps of 128.

2. **Main.c:** In this file, we have included 'matrix.h' so that it takes all the header files and libraries. In this, it computes real or new matrix $C = \text{Alpha} * A * B + \text{Beta} * C$ using block

multiplication algorithm where A, B, and C are matrices and Alpha and Beta are double-precision scalars. We have three matrices:

- A of dimensions $x * y$
- B of dimensions $y * z$
- C of dimensions $x * z$

Therefore, with the help of these matrices, our task is to compute the matrix C, and this main function every time launches different threads from 1 to 4 in steps of 2 and tests the speed of the program by using matrices of dimensions from $960 * 960$ to $1472 * 1472$ in steps of 128. So we exercise the main function in which we have declared the int variable like x, y, z which are dimensions of matrices and we have 'b' which is the block of the matrix as the big matrices are divided in small matrices of size $b * b$ and block algorithm has the advantage of fitting in cache memory. And we have 'bmin' and 'bmax' which are the short form of minimum block size and maximum block size respectively. Next, we have defined here row, col and nth which is the number of threads and these are int variables. Next, we have is an array of A, B, C and D matrix in this A and B are the arrays used to store matrices A and B respectively and C is the array used to store resultant matrix C and lastly D is for dynamic memory allocation. Alpha is the real value used to scale the product of matrices A and B whereas Beta is the real value used to scale the resultant matrix C. The following statements assigns values to the scalars Alpha and Beta then call the function to compute matrix C by multiplying matrix A and B.

```
alpha = 1.0  
beta = 2.0
```

The statement below I have used 'malloc' function which allocates the memory of the size of the matrix to A from the heap (we can use either Stack or Heap memory efficiently, so the stack is fast rather than heap memory, but stack has limited memory). Hence, to preserve large input memories, we use heap memory. And returns the address pointer of memory to matrix A. We used the Maximum dimension of the row and a maximum dimension of the column so that it can allocate maximum dimension value in its array.

```
A = (double *) malloc(sizeof(double) * MAX_D * MAX_D);
```

Now after allocating the memory to the matrices, we then initialize the arrays A, B, and D using a random function and using for loops like in the below code the array A is initialized using the random function or initialized matrix data if we haven't used this random function then it will be hectic or near too impossible to write the array of this much size.

```
for(row=0; row<MAX_D; row++)  
    for(column=0; column<MAX_D; column++)  
        A[row* MAX_D +column] = rand();
```

After initializing the arrays, A, B and D we then use for loop because the initial value is the minimum dimensions and it's in loop for maximum dimensions and incrementing by 128. Since the matrices are square matrices $x = y = z$ so below this we are allocating block size according to number of threads. Here we have taken number of threads as 4. So, if number of threads is 4 such that in this case then block size is between 4 and 64, otherwise block size will be 16. Then we are setting number of OpenMP threads as number of threads to find time 1 and then by using the 'MatMul' function we computes the matrix C so we can also say that time is when we started the computation by setting 'nth' in omp and time 2 is when we end the computation by running the program and then finally difference between time 2 and time 1 is actual time taken by program for multiplication of matrix using blocks. Now in the last section of code we are validating the random entries in the matrices with the help of scalar alpha and beta. Lastly printing dimensions of the matrix x, y, z, the number of threads, the block size and the time taken by program for multiplication of matrix and then finally the status whether its 'Failed' or 'Passed' in the output. Lastly freeing up the memory by using fflush main purpose is to clear or flush the output buffer and move the buffered data to console (by using stdout).

3. **MMultiplication.c:** In this c program we make use of openMP to parallelize the computation of $C = \text{Alpha} * A * B + \text{Beta} * C$. So, what we did in this program is we used for loop to break down the matrix into b blocks and then picking up each $b * b$ block and store this in a 'temp' variable. And finally using block multiplication algorithm just to multiply both the matrices A and B and store it in matrix C and then adding the 'temp' variable scaled by factor of Beta to C matrix. In the below code we are multiplying the blocks of A [i, k] matrix and B [k, j] matrix and storing the output in C [i, j] matrix.

```
for(ii = i; ii < i+b; ii++)
{
    for(jj = j; jj < j+b; jj++)
    {
        for(kk = k; kk < k+b; kk++)
        {
            C[ii*z+jj] += A[ii*y+kk] * B[kk*z+jj];
        }
    }
}
```

And Lastly computing the C matrix as $\text{Alpha} * A * B + \text{Beta} * C$

```
C[ii*z+jj] = Alpha * C[ii*z + jj] + Beta * Ctemp[(ii-i)*b + (jj-j)];
```

4. **Makefile:** The makefile is an exceptionally handy automation tool it just compiles and runs the program more effectively. If you need to run or update an undertaking when certain files are updated, the make utility can prove to be useful. The make utility requires a file, Makefile (or makefile), which defines a set of tasks to be executed. Make is used to compile a program from source code. Most open-source projects use make to compile a final executable binary, which can then be installed using make install.

```

CC=gcc
TARGET=main

CFLAGS = -Wall -fopenmp
all:    main.o MMultiplication.o
$(CC) $(CFLAGS) main.c MMultiplication.c -o $(TARGET)

clean:
rm *.o $(TARGET)

```

So, in this makefile first we must mention the compiler i.e. 'gcc' and then target file name which is 'main' and then using flags as '-fopenmp' and '-Wall' that is to be used with the default compiler.

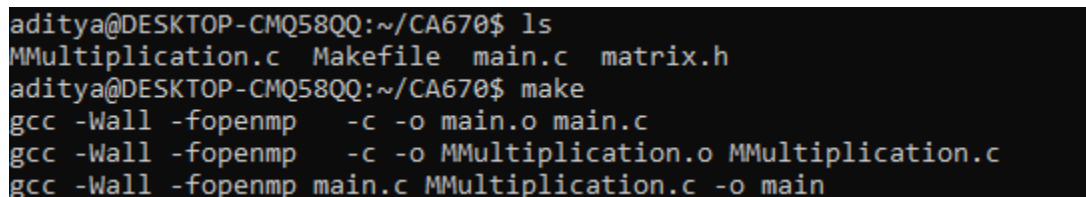
OUTPUT

To compile the program in openMP certain changes had to be made in order to execute the program. I used a Linux machine to compile and link the program. OpenMP is nothing but a library for executing C, C++ program on multiple processors at the same time. It consists of compiler directives, and that is the `#pragma omp parallel`. These are the directives for the compiler. These are picked up by the compiler when we compile the code and replaced with some other code, so the compiler figures out what to put over there. And then are runtime libraries these are functions that can be invoked in the code. OpenMP makes the code much faster if it uses many loops and utilizes the full power of CPU.

To run the program on Ubuntu/Linux

1. Run 'sudo apt-get install libomp-dev' in the terminal.
2. Create a C project and title CA670.
3. Then go to 'Properties', selecting the project.
4. Go to C/C++ Build and select settings.
5. Select GCC compiler.

After executing all the steps above our program is ready to build in openMP. We will then create a directory in home folder CA670 and then we put all the files containing the program and then execute this below command: 'make'



```

aditya@DESKTOP-CMQ58QQ:~/CA670$ ls
MMultiplication.c  Makefile  main.c  matrix.h
aditya@DESKTOP-CMQ58QQ:~/CA670$ make
gcc -Wall -fopenmp -c -o main.o main.c
gcc -Wall -fopenmp -c -o MMultiplication.o MMultiplication.c
gcc -Wall -fopenmp main.o MMultiplication.o -o main

```

Figure 1: Running the command 'make'

After compiling the program, we run './main' file then we'll get this output:

```
aditya@DESKTOP-CMQ58QQ:~/CA670$ ./main
{x=960,y=960,z=960,nth=1,b=16,time=12,Status=Passed}, {x=960,y=960,z=960,nth=2,b=16,time=6,Status=Passed},
{x=960,y=960,z=960,nth=4,b=4,time=4,Status=Passed}, {x=960,y=960,z=960,nth=4,b=16,time=4,Status=Passed},
{x=960,y=960,z=960,nth=4,b=64,time=3,Status=Passed}, {x=1088,y=1088,z=1088,nth=1,b=16,time=18,Status=Passed},
{x=1088,y=1088,z=1088,nth=2,b=16,time=9,Status=Passed}, {x=1088,y=1088,z=1088,nth=4,b=4,time=5,Status=Passed},
{x=1088,y=1088,z=1088,nth=4,b=16,time=6,Status=Passed}, {x=1088,y=1088,z=1088,nth=4,b=64,time=5,Status=Passed},
{x=1216,y=1216,z=1216,nth=1,b=16,time=25,Status=Passed}, {x=1216,y=1216,z=1216,nth=2,b=16,time=13,Status=Passed},
{x=1216,y=1216,z=1216,nth=4,b=4,time=9,Status=Passed}, {x=1216,y=1216,z=1216,nth=4,b=16,time=7,Status=Passed},
{x=1216,y=1216,z=1216,nth=4,b=64,time=7,Status=Passed}, {x=1344,y=1344,z=1344,nth=1,b=16,time=34,Status=Passed},
{x=1344,y=1344,z=1344,nth=2,b=16,time=17,Status=Passed}, {x=1344,y=1344,z=1344,nth=4,b=4,time=13,Status=Passed},
{x=1344,y=1344,z=1344,nth=4,b=16,time=10,Status=Passed}, {x=1344,y=1344,z=1344,nth=4,b=64,time=10,Status=Passed},
{x=1472,y=1472,z=1472,nth=1,b=16,time=46,Status=Passed}, {x=1472,y=1472,z=1472,nth=2,b=16,time=23,Status=Passed},
{x=1472,y=1472,z=1472,nth=4,b=4,time=19,Status=Passed}, {x=1472,y=1472,z=1472,nth=4,b=16,time=12,Status=Passed},
{x=1472,y=1472,z=1472,nth=4,b=64,time=13,Status=Passed}, aditya@DESKTOP-CMQ58QQ:~/CA670$
```

Figure 2: Output of Matrix Multiplication with openMP

So, what does this output show? This output shows the dimensions of x, y and z starting from 960 to 1472 in the sets of 128. The next we can see that number of thread 'nth' is incrementing from 1,2 then to 4 because total number of threads were 4. The 'b' block size is between 4 and 64, otherwise block size will be 16. Next is actual time taken by program for multiplication of matrix using blocks has been shown for different number of threads and for different block size like if we see for the dimension 960 then

```
{x=960,y=960,z=960,nth=1,b=16,time=12,Status=Passed}, {x=960,y=960,z=960,nth=2,b=16,time=6,Status=Passed},
{x=960,y=960,z=960,nth=4,b=4,time=4,Status=Passed}, {x=960,y=960,z=960,nth=4,b=16,time=4,Status=Passed},
{x=960,y=960,z=960,nth=4,b=64,time=3,Status=Passed}, {x=1088,y=1088,z=1088,nth=1,b=16,time=18,Status=Passed}
```

Figure 3: Example of output with dimension 960 * 960 of a matrix.

It can be inferred that when x, y, z dimensions were 960 then like every other dimension it will see the time taken when nth was 1 and b was 16 so the time that it took to multiply was 12 secs and when nth was 2 and b was 16 then the time that it took to multiply was 6 secs when nth was 4 and b was 4 then time it took was 4 secs when nth was 4 and b was 16 then the time that it took was 4 secs and then finally the last thread i.e. 4 and when the block size was 64 then the time that it took was the least that is 3 seconds so it can be seen that when there was the maximum number of threads and when the block size was maximum then the time taken by the program for multiplication of matrix using blocks was found to be the least and whereas when the number of threads was least i.e. 1 and block size was 16 then it took maximum time to multiply the matrix.

So, this was the case when we used #pragma omp parallel basically the "parallel" directives ensure that the two sections are assigned to two different threads. When we run the same code by making changes to MMultiplication.c and removing the #pragma omp parallel and moving the program to another .c file which is MatMulWithoutOpenmp.c then we get this output as below:

```

aditya@DESKTOP-CMQ58QQ:~/CA670$ make
gcc -Wall -fopenmp -c -o MatMulWithoutOpenmp.o MatMulWithoutOpenmp.c
gcc -Wall -fopenmp main.c MatMulWithoutOpenmp.c -o main
aditya@DESKTOP-CMQ58QQ:~/CA670$ ./main
{x=960,y=960,z=960,nth=1,b=16,time=13,Status=Passed}, {x=960,y=960,z=960,nth=2,b=16,time=12,Status=Passed},
{x=960,y=960,z=960,nth=4,b=4,time=16,Status=Passed}, {x=960,y=960,z=960,nth=4,b=16,time=12,Status=Passed},
{x=960,y=960,z=960,nth=4,b=64,time=13,Status=Passed}, {x=1088,y=1088,z=1088,nth=1,b=16,time=19,Status=Pa
ssed}, {x=1088,y=1088,z=1088,nth=2,b=16,time=18,Status=Passed}, {x=1088,y=1088,z=1088,nth=4,b=4,time=21,St
atus=Passed}, {x=1088,y=1088,z=1088,nth=4,b=16,time=19,Status=Passed}, {x=1088,y=1088,z=1088,nth=4,b=64,ti
me=19,Status=Passed}, {x=1216,y=1216,z=1216,nth=1,b=16,time=25,Status=Passed}, {x=1216,y=1216,z=1216,nth=2
,b=16,time=28,Status=Passed}, {x=1216,y=1216,z=1216,nth=4,b=4,time=42,Status=Passed}, {x=1216,y=1216,z=121
6,nth=4,b=16,time=26,Status=Passed}, {x=1216,y=1216,z=1216,nth=4,b=64,time=29,Status=Passed}, {x=1344,y=13
44,z=1344,nth=1,b=16,time=39,Status=Passed}, {x=1344,y=1344,z=1344,nth=2,b=16,time=44,Status=Passed}, {x=1
344,y=1344,z=1344,nth=4,b=4,time=53,Status=Passed}, {x=1344,y=1344,z=1344,nth=4,b=16,time=37,Status=Passed}
, {x=1344,y=1344,z=1344,nth=4,b=64,time=40,Status=Passed}, {x=1472,y=1472,z=1472,nth=1,b=16,time=50,Status
=Passed}, {x=1472,y=1472,z=1472,nth=2,b=16,time=53,Status=Passed}, {x=1472,y=1472,z=1472,nth=4,b=4,time=67
,Status=Passed}, {x=1472,y=1472,z=1472,nth=4,b=16,time=49,Status=Passed}, {x=1472,y=1472,z=1472,nth=4,b=64
,time=60,Status=Passed}, aditya@DESKTOP-CMQ58QQ:~/CA670$

```

Figure 4: Output of Matrix Multiplication without OpenMP

So, when we see this output which is without #pragma omp parallel the time that it takes with every thread and block size is different and sometimes its executing different thread and sometimes its executing different thread and execution time is different in all cases.

Efficiency

The easiest way to create parallelism in OpenMP is to use parallel pragma and a block anteceded by the omp parallel pragma is called a parallel region so it will be executed by newly created team of threads. Here I have launched 4 different threads for matrix multiplication. Internally we have divided the workload in static manner and each multiplication instruction would take same amount of time. Here we are dealing with the dimensions of 960, 1088, 1216, 1344, 1472 in the steps of 128, and workload can be divided equally among threads.

Dimensions of Square Matrix	Number of Threads	Block Size	Without using OpenMP (Secs)	With using #pragma omp parallel (Secs)
960 * 960	Nth = 1	B = 16	13	12
	Nth = 2	B = 16	12	6
	Nth = 4	B = 4	16	4
	Nth = 4	B = 16	12	4
	Nth = 4	B = 64	13	3
1088 * 1088	Nth = 1	B = 16	19	18
	Nth = 2	B = 16	18	9
	Nth = 4	B = 4	21	5
	Nth = 4	B = 16	19	6
	Nth = 4	B = 64	19	5
1216 * 1216	Nth = 1	B = 16	25	25
	Nth = 2	B = 16	28	13
	Nth = 4	B = 4	42	9
	Nth = 4	B = 16	26	7
	Nth = 4	B = 64	29	7

1344 * 1344	Nth = 1	B = 16	39	34
	Nth = 2	B = 16	44	17
	Nth = 4	B = 4	53	13
	Nth = 4	B = 16	37	10
	Nth = 4	B = 64	40	10
1472 * 1472	Nth = 1	B = 16	50	46
	Nth = 2	B = 16	53	23
	Nth = 4	B = 4	67	19
	Nth = 4	B = 16	49	12
	Nth = 4	B = 64	60	13

In the above table the output of large matrix multiplication using block algorithm in OpenMP using C language has been shown on the right-hand side and whereas in the second last column the output of matrix multiplication has been shown but without #pragma omp parallel.

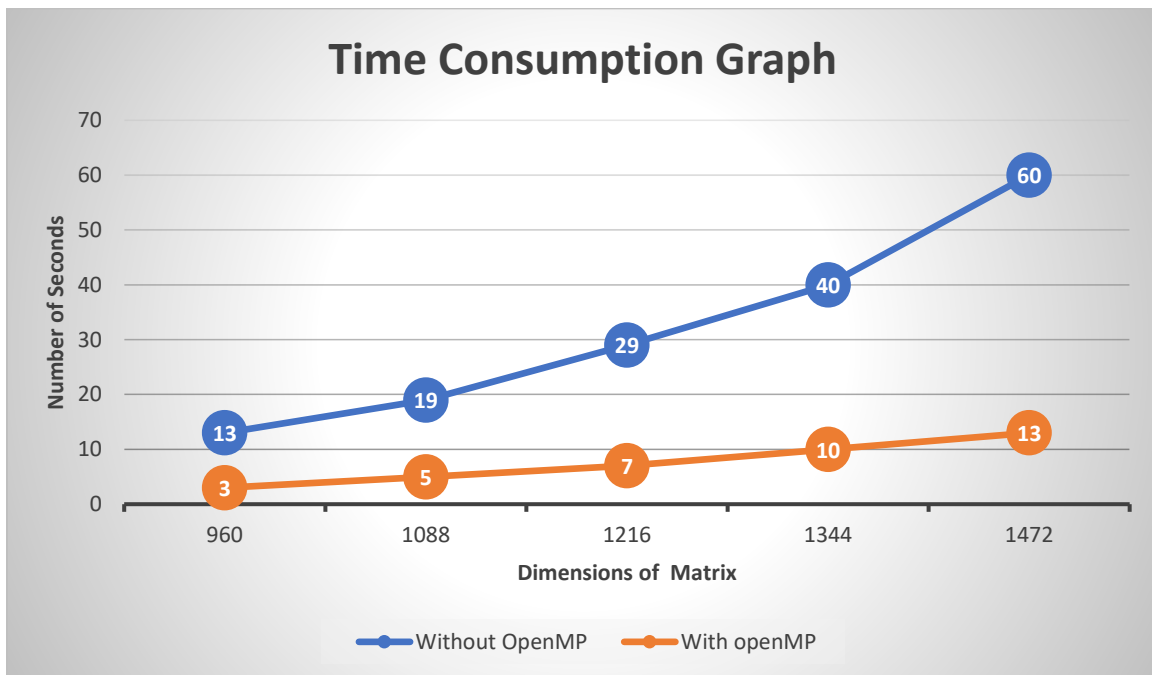


Figure 5: Time Consumption Graph when number of threads = 4 and block size = 64

So, this is the time consumption graph when number of threads were maximum i.e. 4 and the block size was maximum i.e. 64 so it was found out that when we used #pragma omp parallel with the for loop the execution of matrix multiplication was much faster as compared to when there was no use of #pragma omp parallel. The #pragma omp parallel 'for' statement does the loop parallelization by which the matrix can be initialized more efficiently.

References

- [1] <https://software.intel.com/en-us/mkl-tutorial-c-multiplying-matrices-using-dgemm>
- [2] <https://github.com/EvanPurkhiser/CS-Matrix-Multiplication/blob/master/report.md>
- [3] <https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.Examples.pdf>
- [4] <https://opensource.com/article/18/8/what-how-makefile>
- [5] <https://medium.com/swlh/openmp-on-ubuntu-1145355eeb2>
- [6] <https://github.com/sumanthvrao/OpenMP>
- [7] <https://books.google.co.in/books?id=B6XNBQAAQBAJ&pg=PA255&lpg=PA255&dq=alpha+%3D+1.0;+beta+%3D+0.0;+matrix+multiplication&source=bl&ots=t3qHzCZbFG&sig=ACfU3U1WKkOBRjYsjD7V4GmZECCoZO99RA&hl=en&sa=X&ved=2ahUKEwiM0YGHg-roAhVQ73MBHVyQAQYQ6AEwBHoECA0QKQ#v=onepage&q=alpha%20%3D%201.0%3B%20beta%20%3D%200.0%3B%20matrix%20multiplication&f=false>