Efficient Large Matrix Multiplication in OpenMP

1. Assignment Question

We are developing an efficient large matrix multiplication algorithm in OpenMP.

2. Matrix Multiplication

Matrix multiplication is a difficult and time taking task that we have been doing from a very long time. As the size of matrix grows thee time taken to finish the calculation increases likewise. Even computers have the same problem. So, we are developing an an efficient algorithm that takes less time than the conventional method.

If we multiply matrix A & B where n = m = p, the resultant matrix AB is obtained by:

$$\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1p} \\ B_{21} & B_{22} & \cdots & B_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ B_{m1} & B_{m2} & \cdots & B_{mp} \end{pmatrix} \quad (\mathbf{AB})_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj} \, .$$

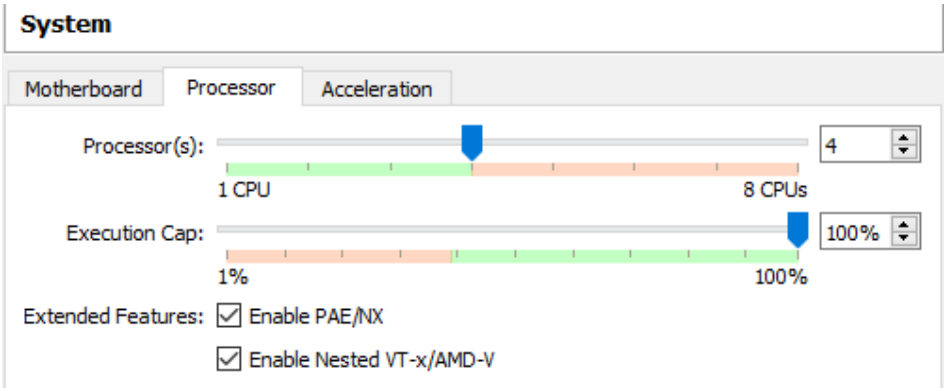
3. System Configuration

Host:

Device specifications

Device name	Raj-1994
Processor	AMD Ryzen 5 3550H with Radeon Vega Mobile Gfx 2.10 GHz
Installed RAM	8.00 GB (7.81 GB usable)
Device ID	[redacted]
Product ID	[redacted]
System type	64-bit operating system, x64-based processor

Virtual Machine:



4. Design of Program

We have integrated parallel block matrix multiplication with OpenMP and simple sequential matrix multiplication in single file and noted the elapsed time for execution for both in different scenarios.

Scenario 1: Noted the elapsed time of execution for block matrix multiplication and sequential matrix multiplication without OpenMP

Scenario 2: Noted the elapsed time of execution for blocks multiplication and sequential multiplication with OpenMP and parallelization implementation and using different #pragma clauses.

Scenario 3: Noted the elapsed time of execution for block and sequential multiplication with OpenMP and increased the num_threads(2) clause of #pragma from 2 to 4 i.e. to num_threads(4) and increased the number of processors from 2 to 4 in the VM. Making the code very efficient and fast.

Below is the detailed explanation of code:-

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

First, we have imported all the required files in the in the program to be used during compilation. #include<omp.h> is an OpenMP library for parallel programming in Symmetric multi-processors. While programming with OpenMP all the threads share memory and data. It has sequential sections and parallel sections. In sequential section it sets up the environment, initializes the variables etc. It has one master thread that runs from beginning to end. When the execution reaches the parallel sections additional threads are forked, these additional threads are slave threads.

#include<time.h> has functions for manipulation date and time.

#include<pthread.h> It allows one to spawn a new concurrent process flow. It is most effective on multi-processor or multi-core systems. It helps gaining speed through parallel or distributed processing.

```
int main(){

#define max_val 10
#define min_val 1

//Initializing Variables
int dimA =0,dimB =0,dimC =0;
int thread_ID, numThreads, i, j, k,N,s,sum=0;
int num_threads =0;
double timetaken = 0, timetaken2 = 0,timetaken2=0;
```

Here we are limiting the min value and the maximum value of every element of the matrix such that its always anywhere between 1 and 10. Then we have initialized all the variables

```
//Input the values of matrix
    printf("Enter the dimension N*N: ");
    scanf("%d" , &N);
    printf("Enter the block_size of Matrix: ");
    scanf("%d" , &s);
```

Here we take in input from the user for the dimension of the matrix and the block size. Here, blocks of particular size is loaded into the cache in order to reduce memory traffic during calculation.

```
//Initialize all the matrix used for Matrix Multiplication and comparison
    int (*matrix1)[N] = malloc(N * sizeof(*matrix1));
    int (*matrix2)[N] = malloc(N * sizeof(*matrix2));
    int (*matrixC)[N] = malloc(N * sizeof(*matrixC));
    int (*matrixOp1)[N] = malloc(N * sizeof(*matrixOp1));
    int (*matrixOp2)[N] = malloc(N * sizeof(*matrixOp2));
```

Here we initialize all the required matrix for Multiplication and Comparison. We have two input matrix matrix1 and matrix2 and two output matrix for sequential solution and parallel OpenMP Solution.

```
//  #pragma omp parallel for schedule (static)
    for(i=0;i<N;i++){
        for(j=0; j<N ;j++){
            matrix1[i][j] = rand() % max_val + min_val;
        }
    }

//  #pragma omp parallel for schedule (static)
    for(i=0;i<N;i++){
        for(j=0; j<N;j++){
            matrix2[i][j] = rand() % max_val + min_val;
        }
```

We add random number from 1 to 10 to the two input matrix matrix1 and matrix2.

```
// OMP Code
struct timeval t0,t1;
gettimeofday(&t0, 0);
```

Gettimeofday(&t0, 0) is a function to obtain time of the system. The time is given in seconds and milliseconds

```
#pragma omp parallel for shared(matrix1,matrix2,matrixOp1,N,s)
private(i,j,k,i1,j1,k1,sum) schedule(auto) num_threads(4) collapse(3) //
```

#pragma omp parallel is used to parallelize the loop. This is the parallel section of the program with multiple threads. This enables a block of code to be executed by multiple threads.
OpenMP Fork-Join Model: Program starts with a single master thread and executes in a sequential order until the first parallel region is encountered.
Fork-Join: Master thread creates number of threads and all the parallel section of the program is executed parallelly. When the parallel execution is done all the threads synchronize and terminate, leaving only the master thread

Here in pragma omp parallel we have used several clauses

- Shared specifies all the variables that are shared among each thread. Private specifies variables that are local to each thread.
- Private specifies variables local to each thread.
- Num_threads specifies number of threads to be created. In our code we have used two different values for Num_threads i.e. 2 threads with 2 cores and 4 threads with 4 core and there was a significant difference in the time taken to execute.
- Schedule describes how iteration of loop are divided among threads. It has different scheduling _calss like static, dynamic, guided, runtime,auto
    - Static: loop iterations are divided into small chunks and then statically assigned to threads
    - Dynamic: When thread finishes one chunk, it is dynamically assigned another.
    - Runtime: The scheduling decision is deferred until runtime.
    - Auto: The scheduling decision is made by compiler.

- Collapse: Specifies how many loops in a nested loop should be collapsed into one large iteration and divided according to schedule clause.

```
for (i=0; i<N; i+=s)
    for (j=0; j<N; j+=s)
      for (k=0; k<N; k+=s)
        for (i1=i;i1<i+s;i1++)
          for (j1=j;j1<j+s;j1++)
            {
              int sum=0;
              for (k1=k;k1<k+s;k1++)
                {
                  sum+=matrix1[i1][k1]*matrix2[k1][j1];
                }
              matrixOp1[i1][j1]+=sum;
            }

gettimeofday(&t1, 0);
timetaken = (t1.tv_sec-t0.tv_sec) * 1.0f + (t1.tv_usec - t0.tv_usec) /
1000000.0f;
```

Parallel Matrix multiplication code with block size implementation, here s is the size of the block given. There matrix will be broken into sections into a collection of small sized matrices and then multiplied. Here the matrices A and B are multiplied together. We also take the time before and after the execution and subtract it to get the elapsed time of the parallel block matrix multiplication.

```
//Serial multiplication
 double matrix_mult_serial(int N);
{
    int i,j,k;
    //double st=omp_get_wtime();
    //DWORD start,end;
    //start = GetTickCount();
    for(i=0;i<N;i++)
    {
        for(j=0;j<N;j++)
        {

            matrixOp2[i][j] = 0.0;


            for(k=0;k<N;k++)
            {
                matrixOp2[i][j] +=  matrix1[i][k]*matrix2[k][j];
            }
        }
    }
```

This is the sequential matrix multiplication code. Here the execution happens in serial mannerfor each element in the matrix without taking performance in consideration. Here too, we capture the time taken for execution.

```
/*
    printf("Result A is : \n");
    for(int i=0;i<N;i++){
        for(int j =0;j<N;j++){
            printf("%d  ",matrix1[i][j]);
        }
        printf("\n");
    }

    printf("Result B is : \n");
    for(int i=0;i<N;i++){
        for(int j =0;j<N;j++){
            printf("%d  ",matrix2[i][j]);
        }
        printf("\n");
    }


    printf("Result C for Parallel : \n");
    for(int i=0;i<N;i++){
        for(int j =0;j<N;j++){
            printf("%d  ",matrixOp1[i][j]);
        }
        printf("\n");
    }
*/

    printf("Elapsed time for OpenMP Parallel: %f \n ", timetaken);
```

Now we print the input matrices and the output matrices for both sequential and parallel implementations with their time taken for execution in different scenarios.

5. Screenshots of Output

```
[root@ca647 Desktop]# gcc -fopenmp multiplication.c -o omp
[root@ca647 Desktop]# ./omp
Enter the dimension N*N:
2
Enter the block_size of Matrix:
2
Result A is :
4       7
8       6
Result B is :
4       6
7       3
Result C for Parallel :
65      45
74      66
Elapsed time for OpenMP Parallel: 0.000121
 Result C for Sequential :
65      45
74      66
Elapsed time for Sequential : 0.000122
```

Output 1: 2X2 matrix multiplication example.

```
[root@ca647 Desktop]# gcc -fopenmp multiplication.c -o omp
[root@ca647 Desktop]# ./omp
Enter the dimension N*N: 3
Enter the block_size of Matrix: 2
Result A is :
4       7       8
6       4       6
7       3       10
Result B is :
2       3       8
1       10      4
7       1       7
Result C for Parallel :
317     90      116
726     64      106
2048    61      138
Elapsed time for OpenMP Parallel: 0.000090
 Result C for Sequential :
71      90      116
58      64      106
87      61      138
Elapsed time for Sequential : 0.000091
```

Output 2: 3X3 matrix multiplication example

2X2 and 3X3 matrices multiplications are done successfully in Output 1 and Output 2 and the results obtained are correct. Now, so that its been verified that the output is correct we comment out the code block which prints the matrices so that we can multiply 2000X2000 matrices and observe the significant difference in the elapsed time taken for execution by parallel and sequential matrix multiplication code blocks in different scenarios.

```
[root@ca647 Desktop]# gcc multiplication.c -o nmp
[root@ca647 Desktop]# ./nmp
Enter the dimension N*N: 2000
Enter the block_size of Matrix: 500
Elapsed time for OpenMP Parallel: 20.150095
 Elapsed time for Sequential : 77.892395
```

Output 3: 2000X2000 matrix multiplication without OpenMP

Here in output 3 we have multiplied 2000X2000 matrices with block size of 500. Here we haven't used OpenMP for parallel programming and commented out #pragma and #include<omp.h>. The parallel code with blocks implementation has executed in 20.150095 seconds and the sequential code took 77.892395 seconds.

```
[root@ca647 Desktop]# gcc -fopenmp multiplication.c -o omp
[root@ca647 Desktop]# ./omp
Enter the dimension N*N: 2000
Enter the block_size of Matrix: 500
Elapsed time for OpenMP Parallel: 11.429518
 Elapsed time for Sequential : 70.077690
```

Output 4: 2000X2000 matrix multiplication with OpenMP and 4 cores with num_threads(2)

Here in Output 3 we have multiplied 2000X2000 matrices with block size of 500 and used OpenMP for parallel programming implementation in the code. Number of cores used were 4 in the Virtual machine and the num_threads clause of #pragma was set to 2. Here we can see significant decrease in the time of execution of codes. The parallel matrix multiplication with blocks has been executed in 11.429518 seconds and sequential matrix multiplication has executed in 70.077690 seconds.

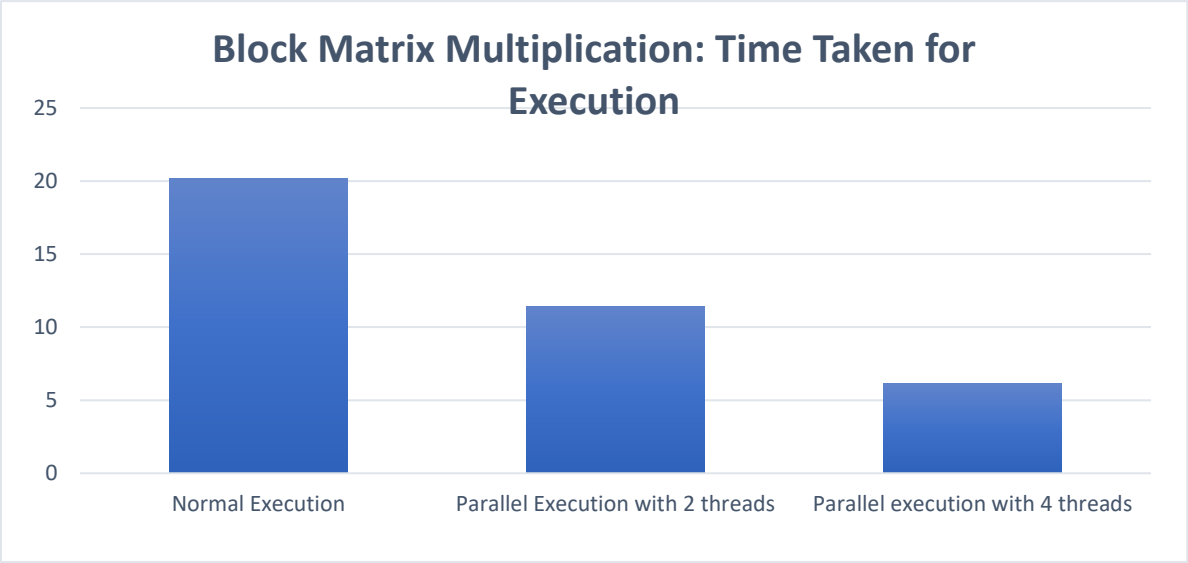
```
[root@ca647 Desktop]# gcc -fopenmp multiplication.c -o omp
[root@ca647 Desktop]# ./omp
Enter the dimension N*N: 2000
Enter the block_size of Matrix: 500
Elapsed time for OpenMP Parallel: 6.155566
 Elapsed time for Sequential : 70.978323
```

Output 5: 2000X2000 matrix multiplication with OpenMP and 4 cores and num_threads(4)

Now we change the #pragma clause num_threads() from 2 to 4 and we can clearly see significant difference in the parallel block matrix multiplication code as the execution time has came down to 6.1555 seconds as there were more threads for the execution, and the sequential matrix multiplication code block took the same time as previous as it had no effect of increasing the number of threads.

6. Results Analysis

We can clearly see that there is a significant decrease in the amount of time taken for the block matrix multiplication code to complete its execution. So by using OpenMP, we used multiple threads to carry out the block matrix multiplication task parallelly, thereby reducing the amount of time required for execution if compared to normal execution.



7. References

[1]https://www.csc.tntech.edu/pdcincs/resources/resources/How%20to%20Design%20a%20Parallel%20Program.pdf

[2]. https://sites.cs.ucsb.edu/~tyang/class/240a17/slides/lectureOpenMP.pdf

[3]. https://www3.nd.edu/~zxu2/acms60212-40212/Lec-12-OpenMP.pdf

[4]. http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

[5]. http://www.bowdoin.edu/~ltoma/teaching/cs3225-GIS/fall17/Lectures/openmp.html