CA670 Concurrent Programming

Name	Rashmiranjan Das
Student Number	19210554
Programme	MCM(Data Analytics)
Module Code	CA670
Assignment Title	OpenMP
Submission Date	17th 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: Rashmiranjan Das Date: 17th April 2020

Open MP
Efficient Large Matrix Multiplication

I. Problem Statement

In this assignment you are to develop an efficient large matrix multiplication algorithm in OpenMP. A prime criterion in the assessment of your assignment will be the efficiency of your implementation and the evidence you present to substantiate your claim that your implementations are efficient.

II Solution

1. Normal sequential Algorithm
2. OpenMP parallel for loop construct
3. Matrix Tiling / Blocking approach
4. Matrix tiling/Blocking with OpenMP parallel for construct

Solution 1: Normal sequential Algorithm

In this Solution the program is implemented in three for loops, it's a dot product of rows in first matrix and columns in second matrix. This algorithm that we use for matrix multiplication is O(n^3) and for each element we perform two operations: multiplication and addition.

```
for ( i = 0; i < n; i++ )
    {
        for ( j = 0; j < n; j++ )
        {
        c[i][j] = 0.0;
           for ( k = 0; k < n; k++ )
           {
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
           }}}
```

This when computed serially/sequentially, takes a long time. Our program is memory bound, which means that the multipliers are not active most of the time because they are waiting for memory. For example, for a matrix size of 500*500, a serial summation takes about 0.577690 seconds. However, when computed parallelly using OpenMP directives, this time reduces.

Solution 2: OpenMP parallel for loop construct

OpenMP parallel for construct was used to parallelize the summation. The parallel construct creates a team of threads which execute in parallel. The variables a, b, c and n are shared between all the threads, while the iterative variables are unique for each thread. The loop construct specifies that the for loop should be executed in parallel. [1]

```
 # pragma omp parallel shared ( a, b, c, n ) private ( i, j, k )
    {# pragma omp for
```

It was observed that, for a typical large array with matrix size equal to 500*500, the time taken to execute the program using a parallel for construct is less than serial execution of the same. For example, for a matrix size of 500*500, a parallel summation with parallel for construct takes about 0.116765 seconds only.

Performance analysis

Matrix size	Sequential approach	Parallel for construct	Execution ratio
100	0.0057875	0.0016744	3.45646
200	0.0329221	0.0112758	2.91971
300	0.0773268	0.0267195	2.89402
400	0.259225	0.0748994	3.46098
500	0.36538	0.116765	3.12918
600	0.706587	0.200326	3.52718
700	1.09159	0.325866	3.34982
800	2.27039	0.595928	3.80985

Solution 3: Matrix Tiling / Blocking approach

Blocking is a technique where we split the large problem into blocks to reduce the working set of data. If we can concentrate on small blocks which fit into the caches, we should get a good speed up.

While performing matrix multiplication using the traditional approach, we run into the issue of Cache pollution where we repeatedly fill the cache with data which we don't completely use.[2]

Diagram 1 below explains the algorithm precisely. [3]

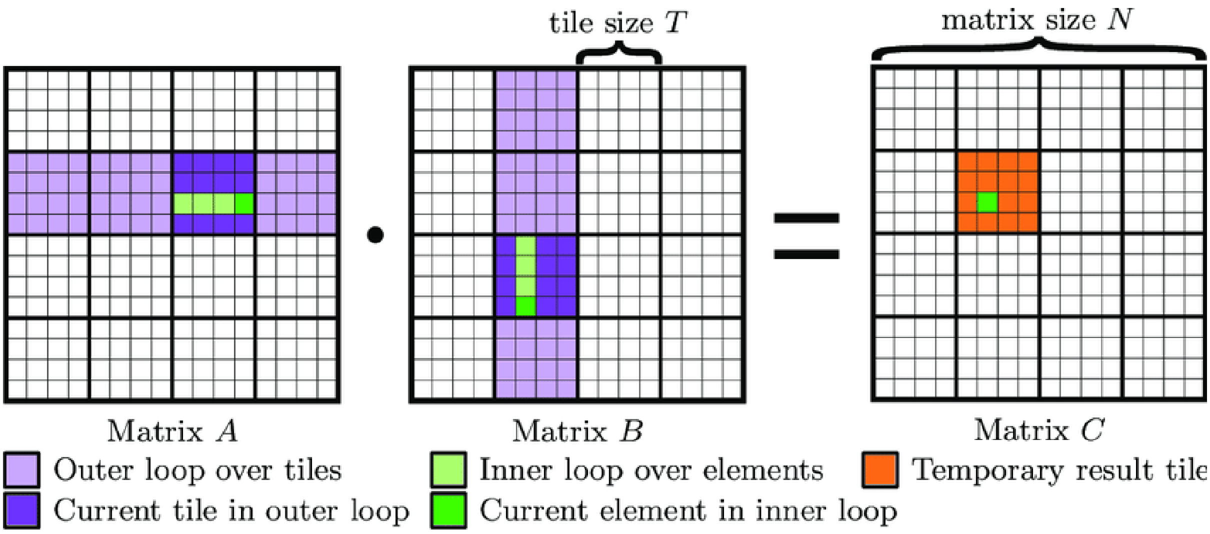

Diagram 1. Block Matrix multiplication implementation

As suggested through the diagram, we can break the problem into blocks. We need to look at specific rows and columns. For the block illustrated above, we need to read the rows 4 to 7 and the columns 4 to 7. If these rows and columns fit into cache, we will get fewer cache misses.

Selecting Block size

Theoretically an optimum block size can be calculated based on the cache size, but it can only be accurately fetched after running through test cases.

Note: Keeping following values constant

*Matrix size: 500*500, Sequential approach time: 0.406973.*

Block Size	Elapsed time	Ratio with naive approach
5	0.418088	0.973414
10	0.364341	1.11701
15	0.356617	1.1412
20	0.352031	1.15607
25	0.333987	1.21853
30	**0.332914**	**1.22246**
35	0.367478	1.10747
40	0.358146	1.13633
45	0.346809	1.17348
50	0.338743	1.20142

As we can see, the optimum time is fetched by block size of 30 from the over all comparison. Hence, we keep 30 as the block size for our block size for all the following test cases.

```
for (int jj = 0; jj < size; jj += block_size)
  {
    for (int kk = 0; kk < size; kk += block_size)
    {
      for (int i = 0; i < size; i++)
      {
        for (int j = jj; j < ((jj + block_size) > size ? size : (jj + block_si
ze)); j++)
        {
          tmp = 0;
          for (int k = kk; k < ((kk + block_size) > size ? size : (kk + block_
size)); k++)
          {
            tmp += a[i][k] * b[k][j];
          }
          c[i][j] += tmp;
        }}}}
```

Performance analysis

As we can see from the test scenarios, the block matrix approach performs better than the traditional sequential approach for 64% of the time.

Size of the Matrix	Block size	Sequential execution	Block Matrix	Execution ratio
50	30	0.0007313	0.0005554	1.31671
100	30	0.0057875	0.0045436	1.27377
150	30	0.0122014	0.0113261	1.07728
200	30	0.0329221	0.0297035	1.10836
250	30	0.0575764	0.0525037	1.09662
300	30	0.0773268	0.0830179	0.93145
350	30	0.122023	0.13532	0.90173
400	30	0.259225	0.246414	1.05199
450	30	0.275967	0.283776	0.97248
500	30	0.36538	0.385784	0.94711
550	30	0.503309	0.516935	0.97364
600	30	0.706587	0.68318	1.03426
700	30	1.09159	1.06741	1.02265
800	30	2.27039	1.83922	1.23444

Solution 4: Matrix tiling with OpenMP parallel for construct

In this approach we combine solution 2 and 3 to achieve the best solution. As we can see the performance improves gradually with the increase in Matrix size.

```
int chunk = 1;
#pragma omp parallel shared(a, b, c, size, chunk) private(i, j, k, jj, kk, tid
, tmp)
    {
    wtime = omp_get_wtime ( );
        #pragma omp for schedule (static, chunk)
```

The schedule(static, chunk-size) clause of the loop construct specifies that the for loop has the static scheduling type. OpenMP divides the iterations into chunks of size chunk-size and it distributes the chunks to threads in a circular order.

Below diagram demonstrates an instance of static scheduling. [4]

```
schedule(static, 4):

****                ****                ****                ****
     ****                ****                ****                ****
          ****                ****                ****                ****
               ****                ****                ****                ****
```

Diagram 2. Static scheduling type with chunk size set to 4

Performance analysis

Matrix size	Block size	Sequential Approach (sec)	OpenMP Parallel for constructs (sec)	Execution ratio	Block matrix (sec)	Execution ratio	Block Matrix parallel Exec (sec)	Execution ratio
50	30	0.0007313	0.0001594	4.58783	0.0005554	1.31671	0.0003475	2.10446
100	30	0.0057875	0.0016744	3.45646	0.0045436	1.27377	0.001749	3.30903
150	30	0.0122014	0.003473	3.51322	0.0113261	1.07728	0.003119	3.91196
200	30	0.0329221	0.0112758	2.91971	0.0297035	1.10836	0.0106327	3.09631
250	30	0.0575764	0.0159685	3.60562	0.0525037	1.09662	0.0135931	4.23571
300	30	0.0773268	0.0267195	2.89402	0.0830179	0.93145	0.0294191	2.62846
350	30	0.122023	0.0399561	3.05392	0.13532	0.90173	0.0382972	3.18621
400	30	0.259225	0.0748994	3.46098	0.246414	1.05199	0.0672922	3.85223
450	30	0.275967	0.0878298	3.14206	0.283776	0.97248	0.0733637	3.76162
500	30	0.36538	0.116765	3.12918	0.385784	0.94711	0.113155	3.22901
550	30	0.503309	0.157005	3.2057	0.516935	0.97364	0.146266	3.44105
600	30	0.706587	0.200326	3.52718	0.68318	1.03426	0.196815	3.59011
700	30	1.09159	0.325866	3.34982	1.06741	1.02265	0.261271	4.178
800	30	2.27039	0.595928	3.80985	1.83922	1.23444	0.492511	4.60983

NOTE: Execution ratio is comparison with respect to sequential approach.

References:

[1] “OpenMP: For.” http://jakascorner.com/blog/2016/05/omp-for.html (accessed Apr. 16, 2020).

[2] “Cache pollution,” *Wikipedia*. Oct. 28, 2018, Accessed: Apr. 16, 2020. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Cache_pollution&oldid=866192301.

[3] “Snapshot.” Accessed: Apr. 16, 2020. [Online]. Available: https://www.researchgate.net/figure/Performance-critical-A-B-part-of-the-GEMM-using-a-tiling-strategy-A-thread-iterates_fig1_320499173.

[4] “OpenMP: For & Scheduling.” Accessed: Apr. 16, 2020. [Online]. Available: http://jakascorner.com/blog/2016/06/omp-for-scheduling.html.