EGRE 531 Multicore and Multithread Programming

Laboratory Number 6

Date: 04/27/18

Luis Barquero

PLEDGE: __________Luis Barquero____________________

"On my honor, I have neither given nor received unauthorized aid on this assignment"

VCU | Virginia Commonwealth University

Introduction:

This lab served to show how different loop orders affects the cache locality and the optimization on improving cache locality. In order to properly understand the effects of loop ordering, a simple matrix multiplication pseudo code was implemented where the i-j-k loop iterators were modified into three different combinations and their times recorded. From there, the loop combination with the best time is chosen to improve through the process of cache blocking, and finally, the same loop combination is parallelized to further improve it.

Lab Content:

Part 1: Part 1 of the lab required taking the following matrix multiplication pseudo code and modify the loop combination, recording its elapsed time, in order to determine which loop combination is the fastest/best:

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

Part 2: After determining that the i-k-j loop combination is the best one, the next step is to improve its time through cache blocking, which is where the data blocks are structured to conveniently fit into a portion of the L1 or L2 cache. For this matrix multiplication, the size of the block was chose to be 10 through trial and error, and because it is a square multiple of the size of the array, 1000 * 1000.

After determining the size of the block, the for loops were modified in the following manner:

```
for (int ii = 0; ii < SIZE; ii = ii + bsize)
{
        //sum = 0;
        for (int jj = 0; jj < SIZE; jj = jj + bsize)
        {
                for (int kk = 0; kk < SIZE; kk = kk + bsize)
                {
                        for (i = ii; i < (ii + bsize); i = i + 1)
                        {
                                for (j = jj; j < (jj + bsize); j = j + 1)
                                {
                                        for (k = kk; k < (kk + bsize); k = k + 1)
                                        {
                                                c[i][j] += a[i][k] * b[k][j];
                                        }
                                }
                        }

                }
        }
}
```

The reason for the structure is because we want to split the whole algorithm into sections, and we want to apply the matrix multiplications into the block size iterations. In this case, the algorithm is applying the matrix multiplication at every 10 block sizes iterations, producing a better result, since the algorithm is performed in sections instead of the whole 1000*1000 elements.

Part 3: The final part is to properly parallelize the loop to further improve the elapsed time. To best way to parallelize it is to have two pragma omp sections: one that will have the matrices as shared variables and the loop iterators as private, and another for a static schedule implementation, since the parallel block will get assigned threads that will each receive the set of loop iterations it is to execute. The following is the parallelization algorithm implemented for the lab:

```
for (int ii = 0; ii < SIZE; ii = ii + bsize)
{
        //sum = 0;
        for (int jj = 0; jj < SIZE; jj = jj + bsize)
        {
                for (int kk = 0; kk < SIZE; kk = kk + bsize)
                {
                        int i = 0;
                        int j = 0;
                        int k = 0;
```

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

                                    for (i = ii; i < (ii + bsize); i = i + 1)
                                    {
                                            for (j = jj; j < (jj + bsize); j = j + 1)
                                            {
                                                    for (k = kk; k < (kk + bsize); k = k + 1)
                                                    {
                                                            c[i][j] += a[i][k] * b[k][j];
                                                    }
                                            }
                                    }
                            }
                    }
            }
    }
```

Test Results:

Part 1: Figure 1 contains the times acquired for each loop combination as it is performed 10 times, in addition to the average times.

	Runs	i-j-k	i-k-j	j-i-k	j-k-i	k-i-j	k-j-i
Time(s)	1	2.793	2.208	2.712	7.29	2.219	7.195
	2	2.815	2.233	2.799	7.343	2.238	7.244
	3	2.802	2.208	2.79	7.341	2.231	7.246
	4	2.764	2.227	2.775	7.337	2.237	7.221
	5	2.795	2.189	2.776	7.276	2.194	7.177
	6	2.799	2.239	2.787	7.339	2.234	7.17
	7	2.745	2.194	2.75	7.263	2.21	7.309
	8	2.73	2.178	2.772	7.31	2.219	7.115
	9	2.767	2.224	2.79	7.302	2.221	7.18
	10	2.73	2.209	2.766	7.284	2.189	7.238
	Average	2.774	2.2109	2.7717	7.3085	2.2192	7.2095

Figure 1 – Figure 1 shows the table containing all 6 i-j-k combinations and their elapsed time.

Using the results, the combinations that perform the best/fastest are the i-k-j and the k-i-j loop combinations, and the reason is because the way cache memory will store information is row-major order, or:
[0][0], [0][1], …, [0][n], [1][0], [1][1], …, [1][n], [m][n].

The matrix multiplication can then be redefined as C[i][j] = A[i][k] * B[k][j].
Given that i appears in the row position twice, this means that to mimic the way cache memory is stored, i has to be the inner loop iterator.

At the same time, the loop iterator k appears once in the row position, which means that to have a faster/better performance, k has to be the second loop iterator.

Finally, since j does not appear in the row position in the algorithm, this means that j has to be the final loop iterator to provide the faster/better performance. Therefore, j-k-i performs better than k-j-i.

Part 2: After choosing the i-k-j loop combination, the loop is improved through cache blocking. Figure 2 shows the improvement in time after 10 runs.

Runs	Cache Blocking
1	2.198
2	2.222
3	2.196
4	2.222
5	2.192
6	2.197
7	2.225
8	2.208
9	2.232
10	2.199
Average	2.2091

Figure 2 – Figure 2 shows the cache block elapsed time results for 10 runs, including the average of the 10 runs.

From Figure 2, it is evident that the time did improve, not immensely, but still enough to make a difference.

Part 3: Finally, Part 3 will take the cache block improvement and parallelize it to improve its elapsed time even further. Figure 3 shows the parallelization results for the parallelization aspect, from threads 2 to 5, which includes 10 runs for each thread and the average for every run.

	Multithreading Threads	2	3	4	5
Trial	1	1.425	1.219	1.173	1.089
	2	1.53	1.255	1.155	1.08
	3	1.524	1.326	1.22	1.085
	4	1.423	1.249	1.218	1.081
	5	1.484	1.281	1.148	1.081
	6	1.429	1.285	1.257	1.08
	7	1.474	1.264	1.144	1.082
	8	1.513	1.234	2.32	1.085
	9	1.482	1.146	1.194	1.079
	10	1.645	1.212	1.151	1.091
	Average	1.4929	1.2471	1.298	1.0833

Figure 3 – Figure 3 shows the parallelization improvement for the matrix multiplication, which includes 10 runs for every thread from 2 to 5, including the average for every run.

<u>**Problems Encountered:**</u>

The main problem encountered was the cache block implementation of the original matrix multiplication code. At first, the elapsed time was not improving, since I was using the wrong cache block size. However, after some trial and error, the cache block size was chosen to 10, and the elapsed time improved, but once again, not immensely.

Appendix A

Lab 6 Source File

```
// ConsoleApplication1.cpp : Defines the entry point for the console application.
//

/*****************************************************************
Programmed by: Luis Barquero
Purpose: Program will perform matrix multiplication on 1000*1000 matrix.
         and will determine the best i-j-k loop combination.
         The program will then improve the elapsed time by performing cache blocking of
         size 10.
         Finally, the elapsed time will be improved even further through parallelization.

*****************************************************************/

#include "stdafx.h"
#include<iostream>
#include<stdio.h>
#include<time.h>
#include<cstdlib>
#include<ctime>
#include<math.h>
#include<omp.h>

using namespace std;
#define SIZE 1000

//#define l1    i
//#define l2    j
//#define l3    k

int a[SIZE][SIZE];
int b[SIZE][SIZE];
int c[SIZE][SIZE];

int main()
{
    int bsize;
    bsize = 20;
    srand(time(NULL));
    int tn = 6; //the number of threads
    omp_set_num_threads(tn); //you can also set the number of threads here besides the
    environment vairalbe

    for (int i = 0; i < SIZE; i = i + 1)
    {
        for (int j = 0; j < SIZE; j = j + 1)
        {
            a[i][j] = 1 + rand() % 999;
            b[i][j] = 1 + rand() % 999;
            c[i][j] = 0;a
        }
    }

    //Implementation for i-j-k

    cout << "Starting..." << endl;
    clock_t start = clock();

    for (int ii = 0; ii < SIZE; ii = ii + bsize)
    {
        //sum = 0;
        for (int jj = 0; jj < SIZE; jj = jj + bsize)
        {
            for (int kk = 0; kk < SIZE; kk = kk + bsize)
            {
                int i = 0;
                int j = 0;
                int k = 0;

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

                        for (i = ii; i < (ii + bsize); i = i + 1)
                        {
                            for (j = jj; j < (jj + bsize); j = j + 1)
                            {
                                for (k = kk; k < (kk + bsize); k = k + 1)
                                {
                                    c[i][j] += a[i][k] * b[k][j];
                                }
                            }
                        }

                }
            }
        }
    }


    clock_t stop = clock();
    double elapsed = (double)(stop - start) * 1000.0 / CLOCKS_PER_SEC;
    cout << "Elapsed time: " << elapsed << endl;
    cout << "\n" << endl;
}
```