Parallel Matrix Multiplication by using Multicore technique

2014190150

Jung Woo Lee (mgp10067)

1. Introduction

Matrix multiplication is one of the most focused mathematical operations nowadays, due to the rise of machine learning techniques. Additionally, calculations such as image processing require matrix multiplication. Although, a matrix multiplication is useful and frequently used, the process of calculation naively takes $O(n^3)$ time complexity which means if the data size is huge, operation time will increase dramatically. To reduce consumed time for matrix multiplication, many techniques such as using transpose of matrix, block matrix multiplication, and algorithms, are suggested. However, using algorithms to reduce operation time had limitations and thus, parallelized calculations with multicore system is applied to overcome limitations.

In this assignment, I will implement a few algorithms to perform matrix multiplication with reduced time and apply multicore parallelization techniques to reduce more time to achieve as fast performance as possible.

2. Implementation

To optimize matrix multiplication for faster performance, I implemented 3 techniques which are 'matrix multiplication with transpose matrix', 'block matrix multiplication with transpose matrix' and the 'Strassen Algorithm'.

A. Naïve Matrix Multiplication

```
void matmul_ref(const int* const matrixA, const int* const matrixB,
                int* const matrixC, const int n) {
  // You can assume matrixC is initialized with zero
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
      for (int k = 0; k < n; k++)
        matrixC[i * n + j] += matrixA[i * n + k] * matrixB[k * n + j];
}
```

Figure 1. naïve Matrix Multiplication

Matrix multiplication requires 3 nested iterations since to calculate 1 element of result matrix, we need to sum up all the multiplications of elements in rows of first matrix and columns of second matrix. Therefore, in the naïve way of calculation, n * n * n iterations are required.

B. Using Transpose Matrix [1]

The technique that uses transpose matrix cannot reduce time complexity. However, applying A x B^T = C to calculate A x B = C reduces actual runtime dramatically.

```
//apply Transpose by changing order. (better cache line)
for (int i = 0; i < n; i++)
  for (int k = 0; k < n; k++){
    //for cache usage
    int matA = matrixA[i * n + k];
    for (int j = 0; j < n; j++)
      matrixC[i * n + j] += matA * matrixB[k * n + j];
  }
```

Figure 2. Transpose Matrix Multiplication

As represented in the figure2, I did not actually transpose the second matrix, but instead, just changed the direction of iteration. Since the purpose of the technique is reducing L1 cache miss, actual transpose of matrix would be meaningless. By changing the order of iteration, cache miss due to approaching elements in same column, different cache line will be minimized. Therefore, overheads to contact memory is reduced.

C. Block Matrix Multiplication [2]

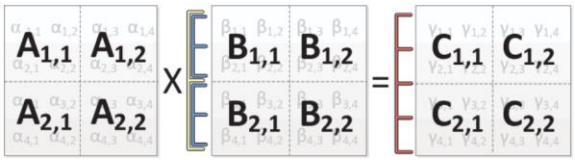

Figure 3. Concept of Block Matrix Multiplication [2]

Block matrix multiplication uses a concept of 'divide and conquer'. This technique also divides a problem into smaller problems to reduce cache miss. Then, by combining results into one solution, the problem can be solved. Moreover, by dividing problems with independent smaller sub problems, multi-threading technique similar to that of reduction can be applied easily. The performance can be improved by applying transpose matrix that was mentioned in *B*.

D. Strassen Algorithm [3]

Strassen Algorithm is also one of divide and conquer algorithm. Unlike other techniques I introduced before, Strassen algorithm reduces time complexity from $O(n^3)$ to $O(n^{2.807})$. Although the difference of number can be considered as tiny number, the performance extremely improves when the dataset is large.

The main concept of Strassen algorithm is dividing matrices into 4 sub-matrices and performing 2 steps of 'tricks'.

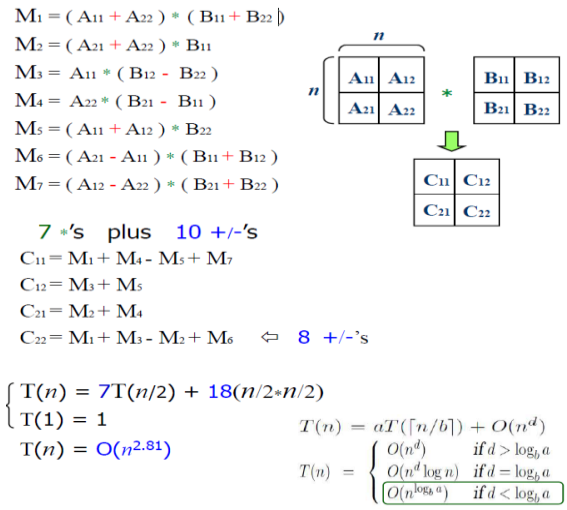

Figure 4. Concept of Strassen Algorithm

In the first step, 7 multiplication and 10 addition/subtractions are used to calculate M values. These M values will be used to calculate final values with only 8 additions/subtractions. By merging the submatrices of solution, we can get final matrix multiplication with reduced multiply operation.

To implement Strassen algorithm, recursive function is required. Also, I used extra functions to add and subtract matrices.

```
47  void Strassen(const int* const matrixA, const int* const matrixB,
48                int* const matrixC, const int n) {
49
50    //apply Strassen Algorithm
51    //threashold of cache = 128;
52    if(n <= 128) {
53        matmul_transpose(matrixA, matrixB, matrixC, n);
54        return;
55    } else {
56
57      int nn = n/2;
58      int* const A11 = (int*)malloc(sizeof(int) * nn * nn);
59      int* const A12 = (int*)malloc(sizeof(int) * nn * nn);
60      int* const A21 = (int*)malloc(sizeof(int) * nn * nn);
61      int* const A22 = (int*)malloc(sizeof(int) * nn * nn);
62      int* const B11 = (int*)malloc(sizeof(int) * nn * nn);
63      int* const B12 = (int*)malloc(sizeof(int) * nn * nn);
64      int* const B21 = (int*)malloc(sizeof(int) * nn * nn);
65      int* const B22 = (int*)malloc(sizeof(int) * nn * nn);
66
67      int i,j,k;
68      for(i = 0; i < nn; i++) {
69        for(j = 0; j < nn; j++) {
70          A11[i*nn + j] = matrixA[i * n + j];
71          A12[i*nn + j] = matrixA[i * n + j + nn];
72          A21[i*nn + j] = matrixA[(i+nn)*n + j];
73          A22[i*nn + j] = matrixA[(i+nn)*n + j + nn];
74          B11[i*nn + j] = matrixB[i * n + j];
75          B12[i*nn + j] = matrixB[i * n + j + nn];
76          B21[i*nn + j] = matrixB[(i+nn)*n + j];
77          B22[i*nn + j] = matrixB[(i+nn)*n + j + nn];
```

Figure 5. Strassen Algorithm (1)

To begin, when recursive function divides the matrices into matrices that has size of less than 128 will perform matrix multiplication with transposed order in parallel by using OpenMP. The threshold of 128 is set according to the server environment to perform. If threshold is 32, matrix multiplication requires 12K memory for 0 capacity misses. Likewise, if threshold is 64, then 48K, if threshold is 128, 192K, and if threshold is 256, 768K is required. The server environment contains L2 cache of 512K, 128 is the most reasonable number for threshold. If we choose 32 since it can perform in L1 cache, division occurs too much which will cause another overhead.

After setting the end point of recursive function, we must allocate memory to hold each sub matrices. Then, copy the values of original matrices to submatrices.

```
81      int*  M1 = (int*)calloc(nn*nn, sizeof(int));
82      int*  M2 = (int*)calloc(nn*nn, sizeof(int));
83      int*  M3 = (int*)calloc(nn*nn, sizeof(int));
84      int*  M4 = (int*)calloc(nn*nn, sizeof(int));
85      int*  M5 = (int*)calloc(nn*nn, sizeof(int));
86      int*  M6 = (int*)calloc(nn*nn, sizeof(int));
87      int*  M7 = (int*)calloc(nn*nn, sizeof(int));
88
89
90      //need to make strassens parallel.
91      #pragma omp task
92        Strassen(mat_add(A11,A22,nn), mat_add(B11,B22,nn), M1, nn);
93      #pragma omp task
94        Strassen(mat_add(A21,A22,nn), B11, M2, nn);
95      #pragma omp task
96        Strassen(A11, mat_sub(B12,B22,nn), M3, nn);
97      #pragma omp task
98        Strassen(A22, mat_sub(B21,B11,nn), M4, nn);
99      #pragma omp task
100       Strassen(mat_add(A11,A12,nn), B22, M5, nn);
101     #pragma omp task
102       Strassen(mat_sub(A21,A11,nn), mat_add(B11,B12,nn), M6, nn);
103     #pragma omp task
104       Strassen(mat_sub(A12,A22,nn), mat_add(B21,B22,nn), M7, nn);
105     #pragma omp taskwait
```

Figure 6. Strassen Algorithm (2)

After dividing all submatrices, calculating M values is the next step. Use calloc to initialize as 0 value and used openMP task to perform Strassen function in parallel. Each function will perform multiplication and save to corresponding M memory. Pragma omp task allow us to use recursive tasks into parallel system. The concept of omp task is similar to thread pool. In matmul_optimize(), single thread put tasks into pool and idle thread will grab task to perform. At last, taskwait performs as barrier to wait all M values calculations are complete. It is important to synchronize the M values since all the M values are required to calculate sub matrices of solution.

```
107     int*  C11 = (int*)malloc(sizeof(int) * nn * nn);
108     int*  C12 = (int*)malloc(sizeof(int) * nn * nn);
109     int*  C21 = (int*)malloc(sizeof(int) * nn * nn);
110     int*  C22 = (int*)malloc(sizeof(int) * nn * nn);
111
112   for(i=0; i<nn*nn; i++) {
113     C11[i] = M1[i] + M4[i] - M5[i] + M7[i];
114     C12[i] = M3[i] + M5[i];
115     C21[i] = M2[i] + M4[i];
116     C22[i] = M1[i] - M2[i] + M3[i] + M6[i];
117   }
118     //merge C
119     for(i = 0; i < nn; i++) {
120       for(j = 0; j < nn; j++) {
121         matrixC[i*n+j] = C11[i*nn+j];
122         matrixC[i*n+j+nn] = C12[i*nn+j];
123         matrixC[(i+nn)*n+j] = C21[i*nn+j];
124         matrixC[(i+nn)*n+j+nn] = C22[i*nn+j];
125       }
126     }
127
128       free(A11);free(A12);free(A21);free(A22);
129       free(B11);free(B12);free(B21);free(B22);
130       free(C11);free(C12);free(C21);free(C22);
131       free(M1);free(M2);free(M3);free(M4);free(M5);free(M6);free(M7);
132       return;
```

Figure 7. Strassen Algorithm (3)

After M value calculation is finished, conquer process is necessary. To maintain cache line locality, operations are performed sequentially.

The disadvantage of Strassen algorithm is that its performance is guaranteed only when the matrices to

multiply have size of exponential of 2's. Therefore, in matmul_optimized(), we must check the size of n, and add sufficient paddings to target matrices to apply Strassen algorithm. Unlike loops in Strassen function, the loops to make paddings can use omp parallel for command. However, loops in Strassen function such as divisions and merge due to the hazard between threads that performs tasks.

3. Experiment & Result

	2048		4096	
	single	multi	single	multi
naïve	INF	INF	INF	INF
transpose	1.93	0.22	18.47	2.28
block&transpose	1.91	0.26	18.41	2.44
strassen	0.99	0.21	7.1	1.35

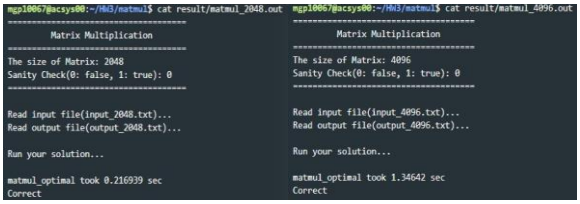

Figure 8. Performance Table(up) Best Performance(down)

4. Analysis

Figure 8 shows the result of best performance and comparisons. Only applying transpose order multiplication showed also showed the dramatic time reduce. However, when the dataset become larger, Strassen algorithm with lower time complexity shows the better performance among all other techniques. Block matrix multiplication showed least performance except naïve calculation since the consideration of cache during development was wrong approach.

Strassen Algorithm showed the best performance. Obviously, lower time complexity will lead the better performance when the data size increases. Also, by parallelize each Strassen recursive function with omp task, the independent calculations will be occurred concurrently.

Moreover, when I used omp loop in Strassen function, the performance took more time since the crashes between threads. Although there were 16 CPUs and 32 threads, using 16 thread showed the maximum performance. In my opinion, although the server can run 32 threads, 16 physical CPU contains 16 L2 Caches and thus using 16 threads with maximum threshold to fit into L2 cache results the best performance.

Additionally, I applied transposed matrix multiplication into Strassen algorithm when the submatrices size meet threshold. The transposed operation also run concurrently by using openMP.

On the other hand, when I reduce threshold from 128, the performance decreased and also, when I increased threshold from 128, performance also decreased. The reason of performance degrade is too much division overhead and the size of L2 cache which is limit due to physical hardware. Furthermore, the thread setting is greater than 16, the performance showed the inverse proportion to the number of threads. This can be another evidence of limits due to physical hardware.

5. Reference

[1] *Martin-thoma, Performance of Matrix multiplication in Python, Java and C++, https://martin-thoma.com/matrix-multiplication-python-java-cpp/*

[2] *Choudhury, A. & Wang, Bei & Rosen, Paul & Pascucci, Valerio. (2012). Topological Analysis and Visualization of Cyclical Behavior in Memory Reference Traces. 9-16. 10.1109/PacificVis.2012.6183557.*

[3] *Kakaradov, Boyko (2004), "Ultra-fast Matrix Multiplication: An Empirical Analysis of Highly Optimized Vector Algorithms", Stanford Undergraduate Research Journal*