

**Advanced Computer Architecture
Programming Homework Report**

**Gaussian Elimination
Multi-thread approach**

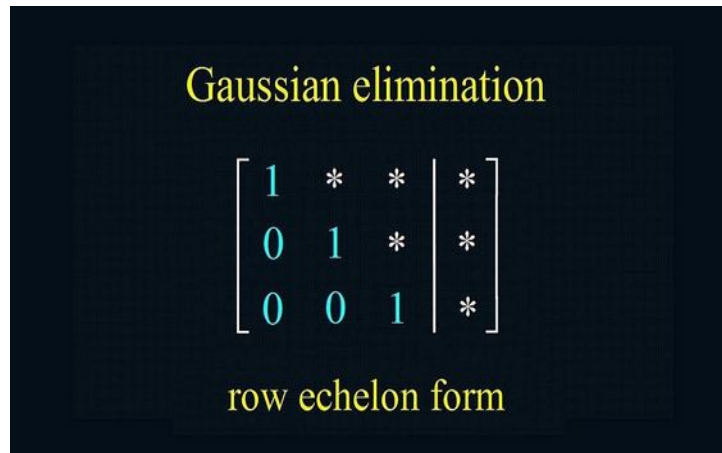
Ali Negary

List of Contents

Introduction to the problem.....	3
Ideas to improve the performance and their challenges.....	4
First attempt:.....	4
Second attempt:.....	4
Third attempt:.....	5
Walk through the code.....	5
Results.....	7
Best Results.....	7
Average Results.....	7

Introduction to the problem

Given in the homework, it is needed to alter a function in the Gaussian Elimination algorithm to enable running in parallel. Gaussian Elimination is a method to solve equations using matrices and linear algebra. This method has two main steps, pivoting and back substitution. In the first step, pivoting, we manage to turn the matrix of coefficients into row echelon form (the following picture):



Gaussian elimination

$$\left[\begin{array}{ccc|c} 1 & * & * & * \\ 0 & 1 & * & * \\ 0 & 0 & 1 & * \end{array} \right]$$

row echelon form

In order to change the matrix into this form, it is required to choose a pivot row in each step and perform addition, subtraction, multiplication and division operations based on that row. For more information you can check out the following links:

- <https://medium.com/linear-algebra/part-6-gaussian-elimination-b1ad4a279a74>
- <https://www.youtube.com/watch?v=2GKESu5atVQ>

According to the pre-implemented code, the time complexity of this algorithm is $O(n^3)$. Despite the challenges, in matrices with large dimensions (1024, 2048, and more) improving timing and performance will be necessary.

Ideas to improve the performance and their challenges

First things to deal with were C programming and Pthread library. For me, the challenge was working with pointers and passing them through threads which took more than how much I estimated to find out. The next step was to do the algorithm on paper and find ways to make the code faster.

```
1 void gauss(double a[][MAX+1], double x[MAX], const int n)
2 {
3     int i,j,k,largest;
4     double t;
5     for(i = 1; i < n; i++){
6         for(largest = i, j = i+1; j <= n; j++)
7             if(abs( a[j][i] ) > abs( a[largest][i] ))
8                 largest = j;
9         for(k = i; k <= n+1; k++)
10             SWAP_DOUBLE( a[largest][k], a[i][k]);
11     for( j = i+1; j <= n; j++)
12         for( k = n+1; k >= i; k--)
13             a[j][k] = a[j][k]-a[i][k]*a[j][i]/a[i][i];
14 }
15 for(i = n; i >= 1; i--){
16     for(t = 0, j=i+1; j <= n; j++)
17         t = t + a[i][j]*x[j];
18     x[i] = (a[i][n+1] - t)/a[i][i];
19 }
20 }
```

If you take a look at the code, you will see 2 main for loops in lines 5 and 15. These are the two main steps of Gaussian Elimination that I mentioned earlier. The first step has 3 nested for loops based on variables n and i. And the second has 2 nested for loops. As I worked on the algorithm, I could not manage to find a solution to improve the second step. In my opinion, this step is difficult to run in parallel. There for, I spent time to find ways to improve the first step which seems to have the main time complexity of the algorithm. Looking at he lines 5 to 13, we have 3 for loops where the third one is a nested for loop. All the loops range from i to n or reverse. Below, I have listed the attempt approaches briefly.

First attempt:

I tried to remove the swap action to simplify the code. This attempt caused failure to find the correct answer.

Second attempt:

In lines 6 to 8, the algorithm iterates through the rows to find the largest value in the column regarding the step of i (i is declared in line 5). Although I tried to make it parallel, it was no good in terms timing and performance. In dimensions of less than 2048, the overhead is more than benefits and in 2048 the overhead is low but honesty it makes no difference. Maybe in larger numbers, ti helps the speed.

Third attempt:

The last approach that I tried and succeeded to improve the performance more than 50%. The improvement was made by breaking lines 11 to 13 into multiple threads.

Walk through the code

First, we need to define the requirements.

```
39  #include <pthread.h>
40
41  #define SWAP_DOUBLE(x, y){double t; t = x; x = y; y = t;}
42  #define MAX 2049
43  #define Threshold 4700
44
45  void readArray2(double[][MAX + 1], const int);
46
47  void gauss(double[][MAX + 1], double[], const int);
48
49  void printArray2(double[], const int);
50
51  void *pivotRoutine(void *arguments);
```

Line 39: Pthread library.

Line 43: Define the variable that will be used in multi-threading process.

Line 51: Calling the function in order to prevent compile errors.

The we need to define a struct to pass between the main Gauss function and the threads. In this struct, I defined 4 integers to record the step of code (baseIndex), the running thread's number or ID (threadIndex), and the row limit in which the thread makes the changes and computations (lowerBound and upperBound).

```
95  typedef struct {
96      int baseIndex;
97      int threadIndex;
98      int lowerBound;
99      int upperBound;
100 } pivotRoutineArguments;
```

Next, we need to define the function which is used in each thread:

```
171 void *pivotRoutine(void *arguments) {
172     pivotRoutineArguments *p = (pivotRoutineArguments *) arguments;
173     int i = p->baseIndex;
174     int lowerRowLimit = p->lowerBound;
175     int upperRowLimit = p->upperBound;
176     int j, k;
177     for (j = lowerRowLimit; j <= upperRowLimit; j++) {
178         double ratio = a[j][i] / a[i][i];
179         for (k = n + 1; k >= i; k--)
180             a[j][k] = a[j][k] - a[i][k] * ratio;
181     }
182 }
```

The following pictures show the updated gauss() function:

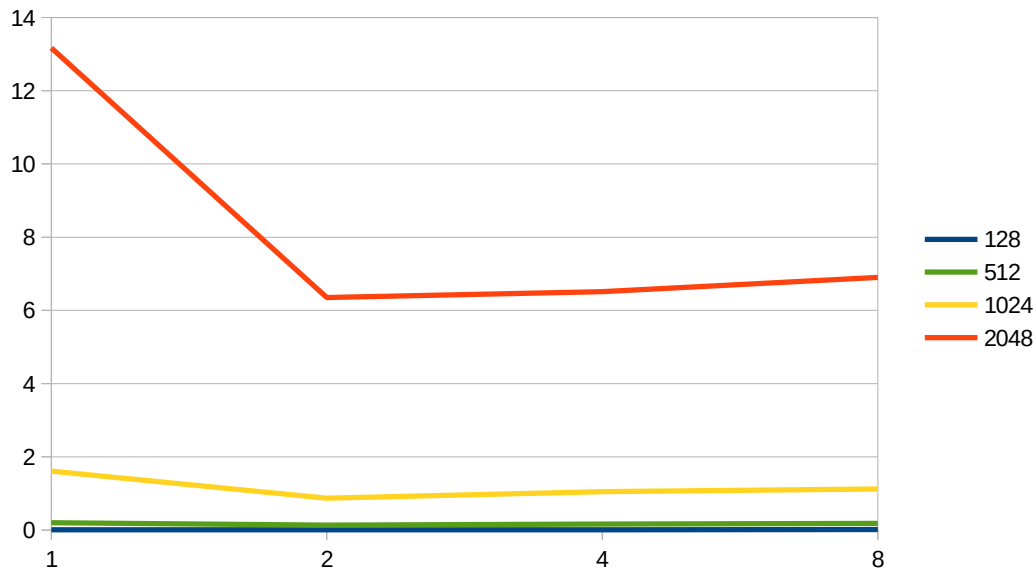
```
111     int my_threads = (n-i) * (n-i) / Threshold;
112     if(my_threads < 1) my_threads = 1;
113     else if (my_threads > threads) my_threads = threads;
114     if (my_threads > 1) {
115         pthread_t thread_id[my_threads];
116         int partition_size = (n - i) / my_threads;
117         int remainder = (n - i) % my_threads;
118         int threadIndex;
119         int result;
120         int new_partition_size;
121         pivotRoutineArguments thread_arguments[my_threads];
122         int lowerBound = i + 1;
123         for (threadIndex = 0; threadIndex < my_threads; threadIndex++) {
124             new_partition_size = partition_size;
125             if (remainder != 0) {
126                 new_partition_size = partition_size + 1;
127                 remainder--;
128             }
129             thread_arguments[threadIndex] = (pivotRoutineArguments) {
130                 .threadIndex = threadIndex,
131                 .baseIndex = i,
132                 .lowerBound = lowerBound,
133                 .upperBound = lowerBound + new_partition_size - 1
134             };
135             result = pthread_create(&thread_id[threadIndex], NULL, pivotRoutine, &thread_arguments[threadIndex]);
136             assert(!result);
137             lowerBound += new_partition_size;
138         }
139         for (threadIndex = 0; threadIndex < my_threads; threadIndex++) {
140             result = pthread_join(thread_id[threadIndex], NULL);
141             assert(!result);
142         }
143     } else {
144         for (j = i + 1; j <= n; j++) {
145             double ratio = a[j][i] / a[i][i];
146             for (k = n + 1; k >= i; k--)
147                 a[j][k] = a[j][k] - a[i][k] * ratio;
148         }
149     }
150 }
```

What happens in this part is making a dynamic multi-threading. By dynamic, I mean when you choose to run the code with for example 8 threads, at one point in the process, the number of threads will change to a lower number. Before implementing this way, I was able to reach 40% improvement in time in 2048 test case. After using this dynamic method, I was able to reach over 50% improvement in performance. For each choice in number of threads, I ran the code 20 times and computed the average time of each choice. Average run-time with single thread is 13.37 seconds. In multi-threading, average best time is 6.45 seconds where best case was 6.18 seconds on 4 threads. All numbers are based on 2048 test case.

In this approach the variable Threshold is used as a decisive point of where to use single threading and where to use multi-threading. The value of 4700 is based on more than 200 retries and observation.

Results

Average results for dynamic version of code are in the following chart:



Best Results

Improvement according to best results for case of 2048:

Number of threads	1	2	4	8	16	32
run-time	11.9	6.18	6.55	6.77	7.32	7.88
Speed up	0	49%	46%	42%	39%	34%

Average Results

Improvement according to average results for case of 2048:

Number of threads	1	2	4	8	16	32
run-time	13.17	6.35	6.61	6.9	7.59	8.34
Speed up	0	53.6%	51.7%	49.6%	44.5%	39.1%

It seems like for this matrix size (2048), most efficient number of threads are 2 and 4.