# Project1 Report

--Shaowei Su

1. Sequential Gaussian Elimination

Gaussian elimination is composed of two major parts: in the first place, the program will transfer the original matrix into row-echelon form by iterating each row i from 0 to size-1, exchanging rows to move the entry with the first non-zero value to the pivot position, eliminating corresponding rows after row I; and the second part is to solve the equation one by one from the bottom.

2. Parallel strategies

As discussed in the requirement, the first part of Gaussian elimination is perfect for parallelism and here I have tried two possible solutions to deal with it. The major part of computeGaussian() includes three for loop indexed by i, j and k correspondingly. As we can see, the inner loop j and loop k is completely independent which could be paralleled in either row oriented view or column oriented view.

2.1 Row oriented approach

Through the outer i loop, certain thread will be responsible to handle rows to find its pivot row. And after that, all the other threads will be launched to reduce the portions of sub matrix they control.

The major subtask function goes like this:

```
void *work_thread(void *id){
    int i,j,k;
    double pivotVal;
    double hrtime1, hrtime2;
    int task_id = *((int *) id);


    barrier(task_num); //wait for all threads to come and then start
    if(task_id == 0){
        hrtime1 = gethrtime_x86();
    }

    for(i=0; i<nsize; i++){
        if(task_id == i % task_num){
            getPivot(nsize,i); // select corresponding thread to find pivot in row
        }
        barrier(task_num); //wait for all threads finish
        pivotVal = matrix[i][i];

        for (j = i + 1 ; j < nsize; j++){

            if(task_id == j % task_num){
                pivotVal = matrix[j][i];
                matrix[j][i] = 0.0;
                for (k = i + 1 ; k < nsize; k++){
                    matrix[j][k] -= pivotVal * matrix[i][k];
                }
                B[j] -= pivotVal * B[i];
            }
        }
        barrier(task_num);
    }
    hrtime2 = gethrtime_x86();

    if(task_id==0){
        printf("Hrtime = %f seconds\n", hrtime2 - hrtime1);
    }
    return NULL;
}
```

Each thread will be responsible for certain rows and it keeps the character of locality.

2.2 Column oriented approach

On the other hand, instead of eliminating row by row, this procedure could be conducted in a column view:

```
for(i=0; i<nsize; i++){
    if(task_id == i % task_num){
        getPivot(nsize,i); // select corresponding thread to find pivot in row
    }
    barrier(task_num); //wait for all threads finish
    pivotVal = matrix[i][i];

    for (j = i + 1; j < nsize; j++){

        if(task_id == j % task_num){
            pivotVal = matrix[i][j];
            //matrix[j][i] = 0.0;
            for (k = i + 1 ; k < nsize; k++){
                matrix[k][j] -= pivotVal * matrix[k][i];
            }
        }
    }
    barrier(task_num);

    if(task_id == 0){
        pivotVal = B[i];
        for(j = i + 1; j < nsize; j++){
            B[j] -= pivotVal * matrix[j][i];
            matrix[j][i] = 0;
        }
    }
    barrier(task_num);
}
```

After finding the pivot row, all threads will be started to partially diminish values below the pivot row.

3. Experimental results

3.1 Environment specification

All of the tests are finished on CSUG nodes cycel2 and cycle3: cycle2 a Dual 6 core CPU with Hyperthreading enabled(that is, the maximum number of logical processors are 24), while cycle3 is which is a Dual 8 core CPU with Hyperthreading enabled(that is, the maximum number of logical processors are 32). Specifically, on Cycle2, the processor Mhz is 2792.994Hz, cache size is 12288kB, and the total memory is 24678888kB; on the other hand, for Cycle3, the processor Mhz is 1200Hz, cache size is 20480kB and total memory is 16382652kB.

To compile the c file:

Gcc pth-gauss1.c hrtimer_x86.c –lpthread

And the input data format is:

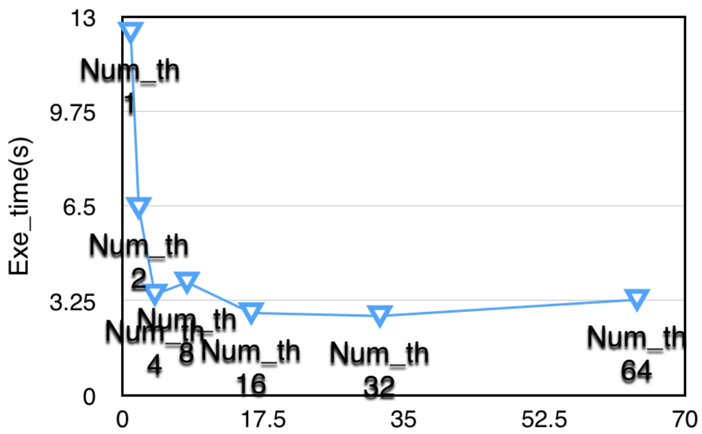
./a.out –s2048 –p32

3.2 Results for Row Oriented Method

3.2.1 Cycle3

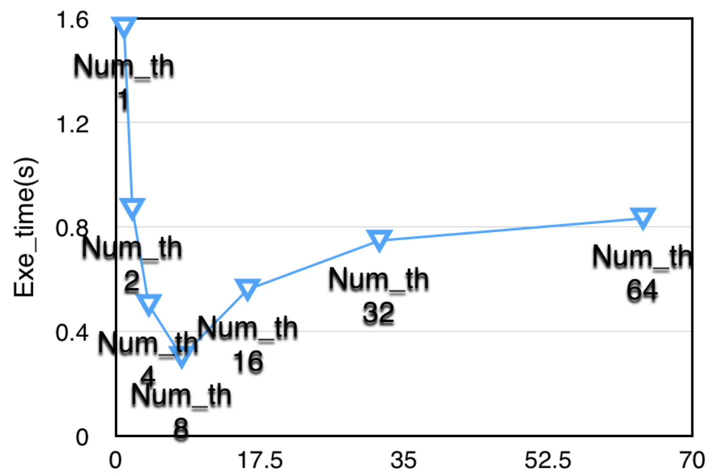
a) s = 2048

| Num_th | Exe_time(s) |
|---|---|
| 1 | 12.4579 |
| 2 | 6.4395 |
| 4 | 3.4615 |
| 8 | 3.8771 |
| 16 | 2.8249 |
| 32 | 2.7215 |
| 64 | 3.2815 |



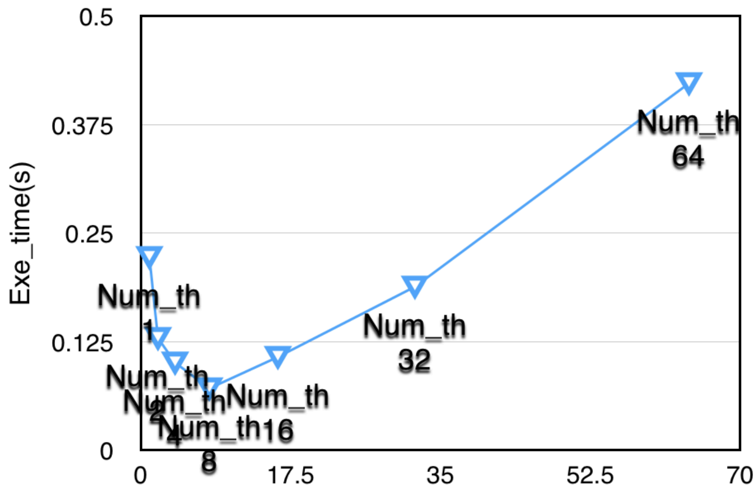

b) s = 1024

| Num_th | Exe_time(s) |
|---|---|
| 1 | 1.5644 |
| 2 | 0.8696 |
| 4 | 0.5014 |
| 8 | 0.3023 |
| 16 | 0.5603 |
| 32 | 0.7479 |
| 64 | 0.8329 |



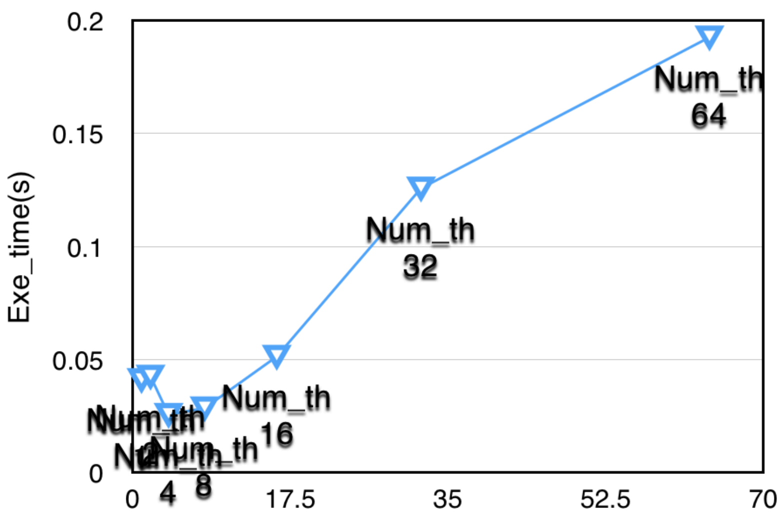

c) s = 512

| Num_th | Exe_time(s) |
|---|---|
| 1 | 0.2224 |
| 2 | 0.1292 |
| 4 | 0.1009 |
| 8 | 0.0712 |
| 16 | 0.1074 |
| 32 | 0.1883 |
| 64 | 0.4226 |



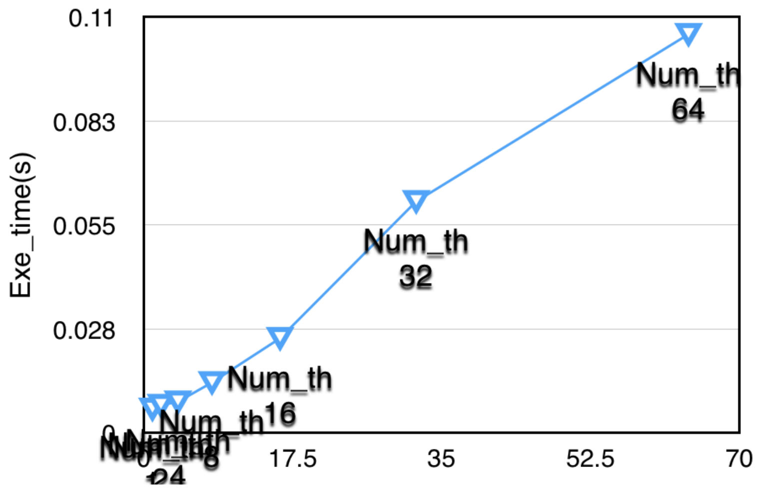

d) s = 256

| Num_th | Exe_time(s) |
|---:|---:|
| 1 | 0.0410 |
| 2 | 0.0425 |
| 4 | 0.0256 |
| 8 | 0.0285 |
| 16 | 0.0513 |
| 32 | 0.1257 |
| 64 | 0.1925 |




e) s = 128

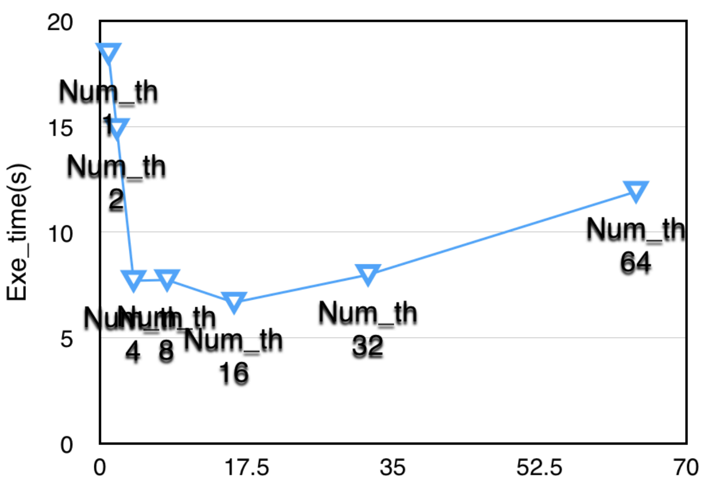
| Num_th | Exe_time(s) |
|---:|---:|
| 1 | 0.0063 |
| 2 | 0.0073 |
| 4 | 0.0082 |
| 8 | 0.0133 |
| 16 | 0.0250 |
| 32 | 0.0612 |
| 64 | 0.1054 |




3.2.2 Cycle2

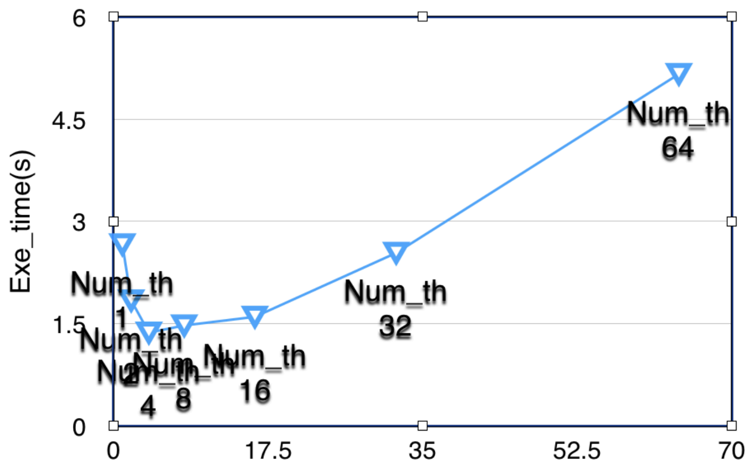
a) s = 2048

| Num_th | Exe_time(s) |
|---:|---:|
| 1 | 18.4631 |
| 2 | 14.8976 |
| 4 | 7.6991 |
| 8 | 7.7360 |
| 16 | 6.6679 |
| 32 | 7.9804 |
| 64 | 11.9222 |



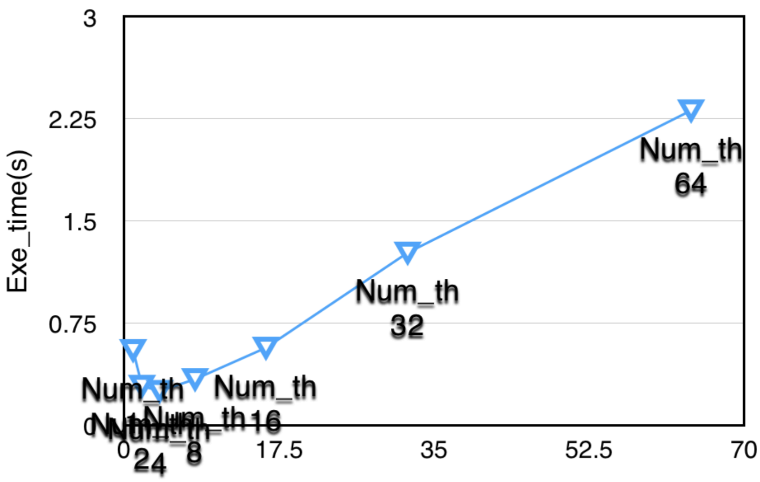
b) s = 1024

| Num_th | Exe_time(s) |
|---|---|
| 1 | 2.6624 |
| 2 | 1.8401 |
| 4 | 1.3681 |
| 8 | 1.4729 |
| 16 | 1.5996 |
| 32 | 2.5379 |
| 64 | 5.1616 |



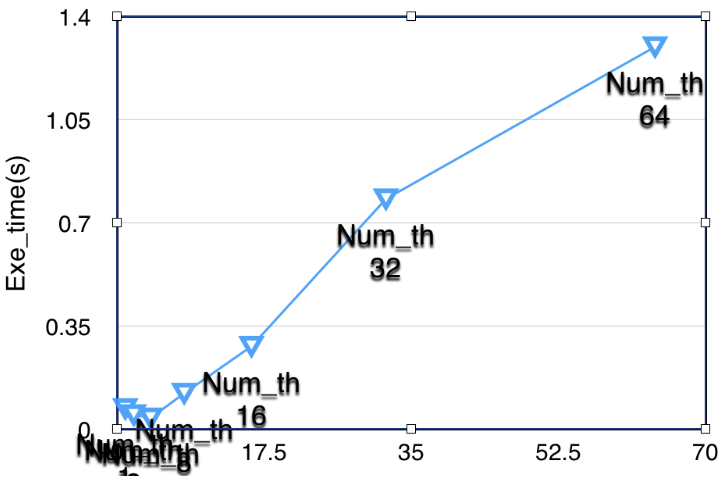

c) s = 512

| Num_th | Exe_time(s) |
|---|---|
| 1 | 0.5511 |
| 2 | 0.2889 |
| 4 | 0.2509 |
| 8 | 0.3381 |
| 16 | 0.5687 |
| 32 | 1.2668 |
| 64 | 2.3112 |



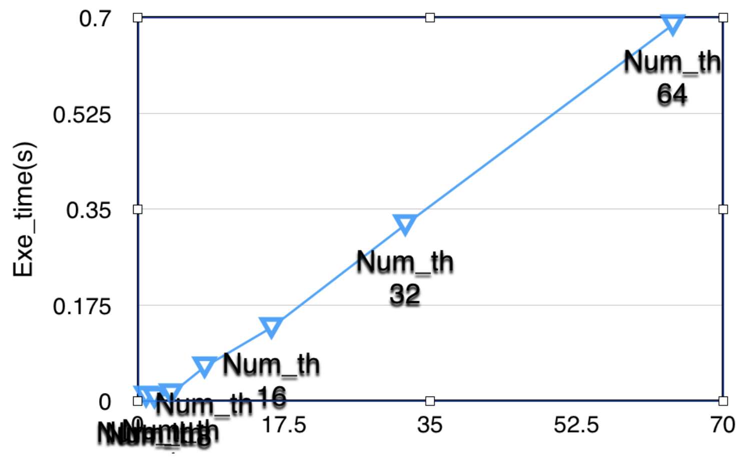

d) s = 256

| Num_th | Exe_time(s) |
|---|---|
| 1 | 0.0695 |
| 2 | 0.0488 |
| 4 | 0.0377 |
| 8 | 0.1217 |
| 16 | 0.2811 |
| 32 | 0.7811 |
| 64 | 1.2971 |



e) s = 128

| Num_th | Exe_time(s) |
|---|---|
| 1 | 0.0086 |
| 2 | 0.0067 |
| 4 | 0.0129 |
| 8 | 0.0620 |
| 16 | 0.1338 |
| 32 | 0.3215 |
| 64 | 0.6866 |

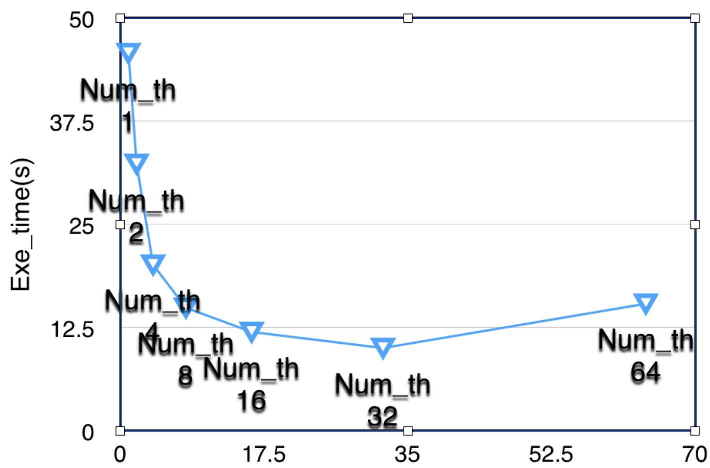



## 3.3 Results for Column Oriented Method

### 3.3.1 Cycle3

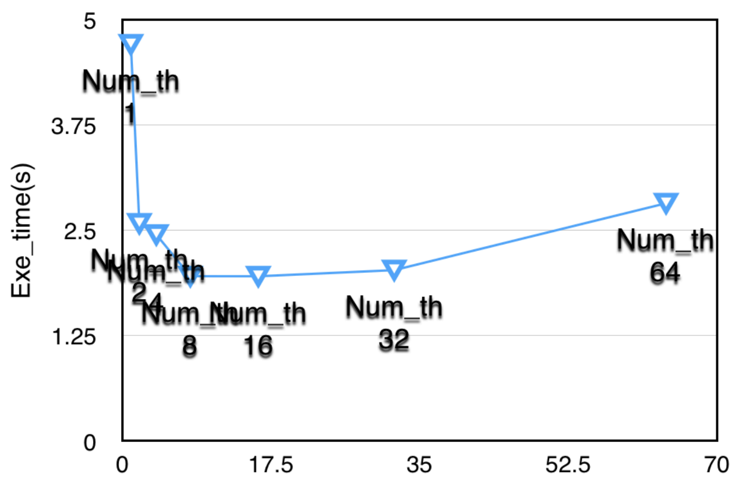
a) s = 2048

| Num_th | Exe_time(s) |
|---|---|
| 1 | 45.7593 |
| 2 | 32.3385 |
| 4 | 20.1281 |
| 8 | 14.9150 |
| 16 | 11.9805 |
| 32 | 10.0381 |
| 64 | 15.4013 |



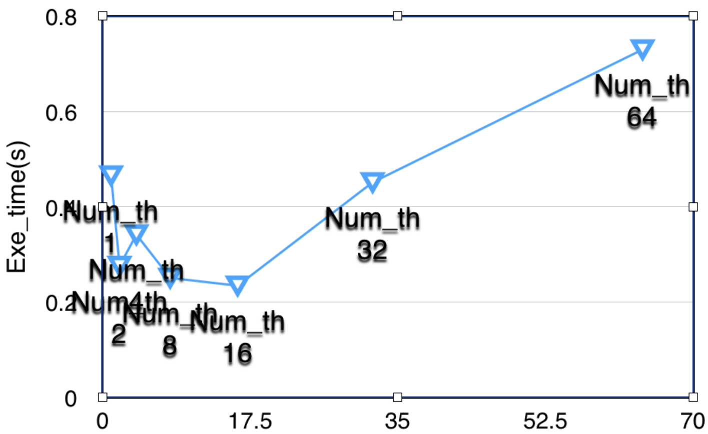

b) s = 1024

| Num_th | Exe_time(s) |
|---|---|
| 1 | 4.7099 |
| 2 | 2.5834 |
| 4 | 2.4504 |
| 8 | 1.9569 |
| 16 | 1.9561 |
| 32 | 2.0291 |
| 64 | 2.8229 |



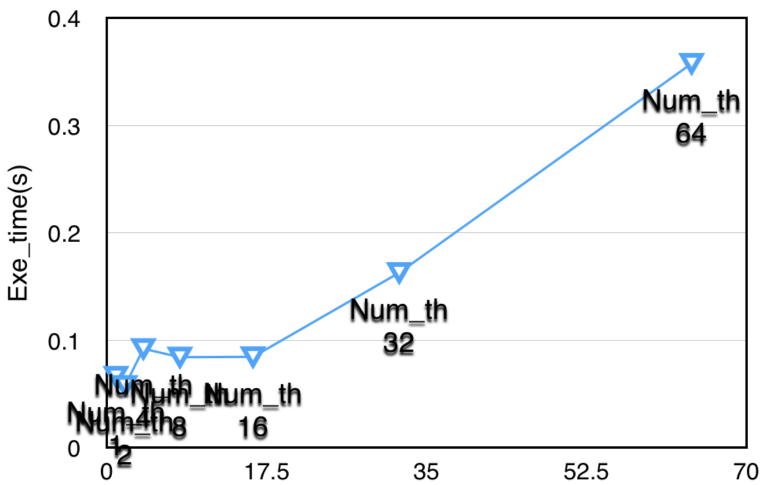

c) s = 512

| Num_th | Exe_time(s) |
|---|---|
| 1 | 0.4657 |
| 2 | 0.2760 |
| 4 | 0.3418 |
| 8 | 0.2509 |
| 16 | 0.2346 |
| 32 | 0.4514 |
| 64 | 0.7300 |



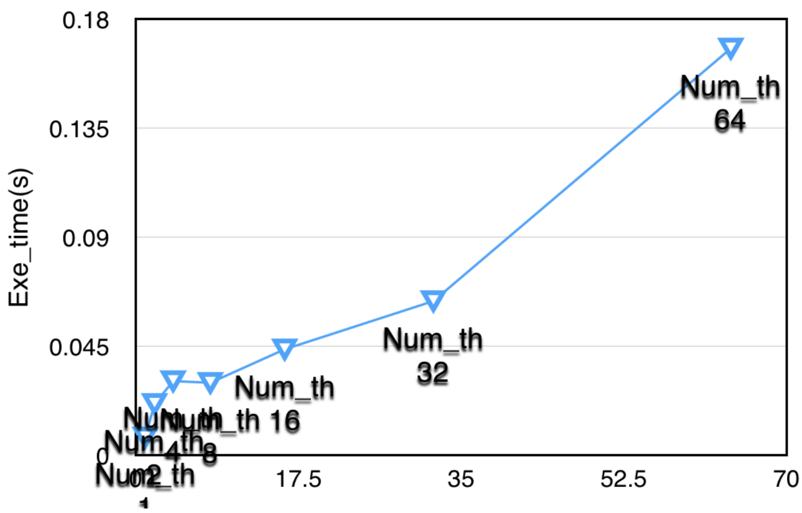

d) s = 256

| Num_th | Exe_time(s) |
|---|---|
| 1 | 0.0663 |
| 2 | 0.0577 |
| 4 | 0.0921 |
| 8 | 0.0842 |
| 16 | 0.0845 |
| 32 | 0.1629 |
| 64 | 0.3577 |




e) s = 128

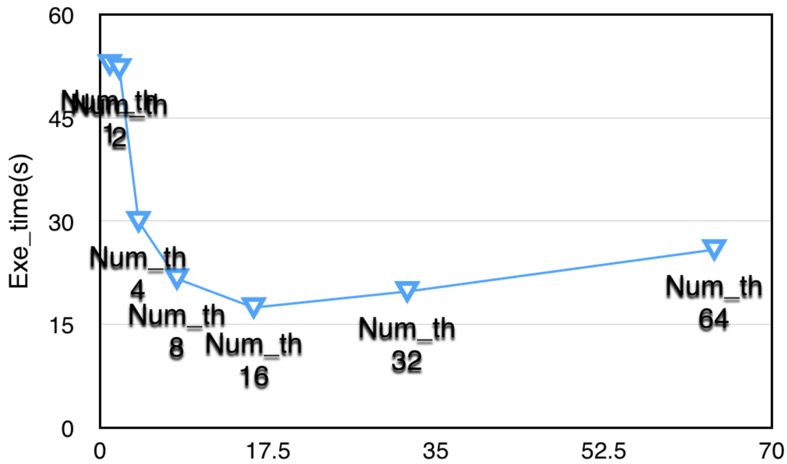
| Num_th | Exe_time(s) |
|---|---|
| 1 | 0.0073 |
| 2 | 0.0211 |
| 4 | 0.0305 |
| 8 | 0.0298 |
| 16 | 0.0436 |
| 32 | 0.0635 |
| 64 | 0.1677 |



3.3.2 Cycle2

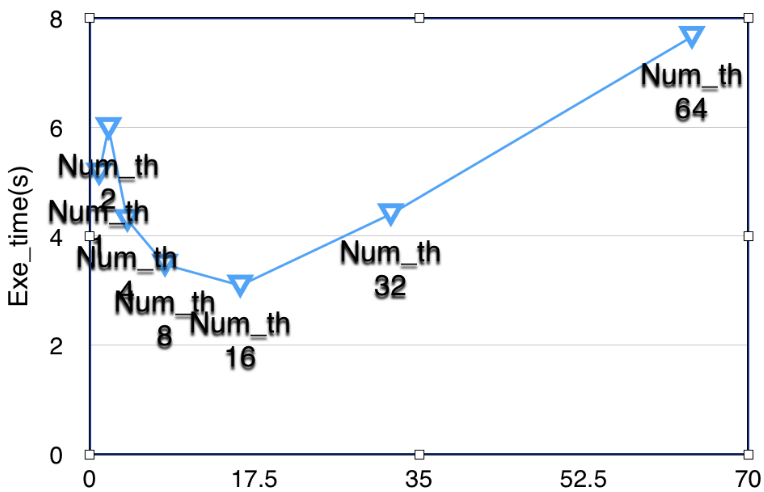
a) s = 2048

| Num_th | Exe_time(s) |
|---|---|
| 1 | 52.7663 |
| 2 | 52.1706 |
| 4 | 29.9635 |
| 8 | 21.6317 |
| 16 | 17.4418 |
| 32 | 19.7752 |
| 64 | 25.8588 |



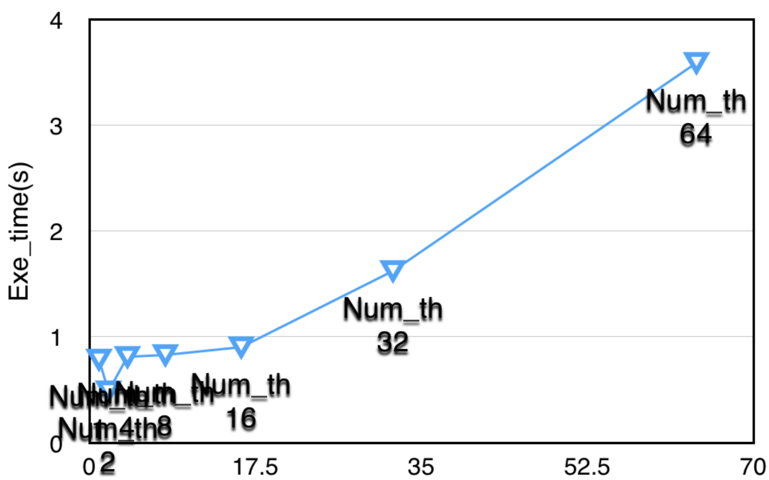

b) s = 1024

| Num_th | Exe_time(s) |
|---|---|
| 1 | 5.1507 |
| 2 | 5.9826 |
| 4 | 4.2976 |
| 8 | 3.4784 |
| 16 | 3.0960 |
| 32 | 4.4038 |
| 64 | 7.6552 |



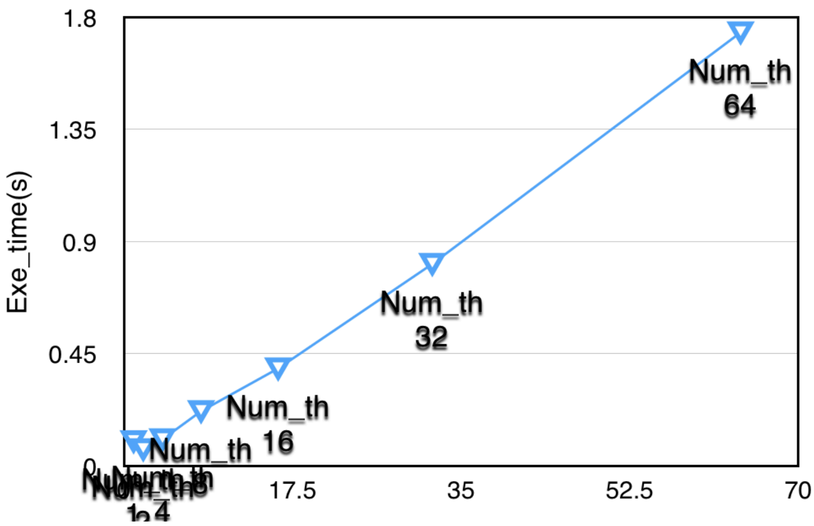

c) s = 512

| Num_th | Exe_time(s) |
|---|---|
| 1 | 0.7863 |
| 2 | 0.4868 |
| 4 | 0.8116 |
| 8 | 0.8289 |
| 16 | 0.9032 |
| 32 | 1.6225 |
| 64 | 3.5885 |



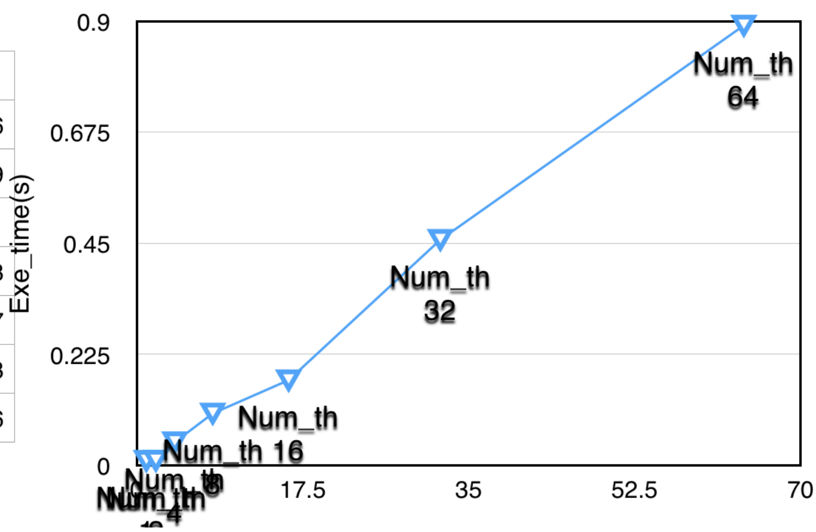
d) s =256

| Num_th | Exe_time(s) |
|---|---|
| 1 | 0.0989 |
| 2 | 0.0665 |
| 4 | 0.1065 |
| 8 | 0.2193 |
| 16 | 0.3906 |
| 32 | 0.8096 |
| 64 | 1.7380 |




e) s = 128

| Num_th | Exe_time(s) |
|---|---|
| 1 | 0.0096 |
| 2 | 0.0099 |
| 4 | 0.0471 |
| 8 | 0.1053 |
| 16 | 0.1727 |
| 32 | 0.4573 |
| 64 | 0.8916 |




3.4 Time analysis

Generally, the execution time is composed by computation time, idle time and communication time. First of all, computation time is the time spent on data computation and it is the part that benefits the most from parallelism. For large size problem, most of time spent on computation; while idle time is the time spent waiting for data from other processor, like I/O for every thread; and finally, communication is the part of time for data transfer between threads. For small size problem, idle time and communication time are the majority part.

3.5 Conclusion

i. The entire execution time heavily depends on the problem size, i.e. matrix length;
ii. Parallelism for small size problem is not always a good idea, since the overheads for threads creation/join and communication count for a large part of time. That is, speed up and efficiency become more ideal when the length of matrix increases;

iii. Mostly influenced by the matter of locality, Column Oriented Method suffers a lot from the I/O issues and turns out to be a inferior solution;

iv. With large problem size(s=1024 or s=2048), increasing the number of threads always leads to better performance and it reaches the best state when logical threads equals to number of physical processors.

4. Extra test case

It is important to check the correctness other than the static initialization, so I add one test case inside:

```
#ifdef CHECK_CORRECTNESS
    nsize = 3;
    allocate_memory(nsize);
    initCheck(nsize);
#else
    allocate_memory(nsize);
    initMatrix(nsize);
#endif
```

```
/*
    test case:
    | 2   1  -1 |  8 |
    | -3 -1   2 | -11|
    |-2   1   2 | -3 |

*/

void initCheck(int nsize)
{
    matrix[0][0] = 2;
    matrix[0][1] = 1;
    matrix[0][2] = -1;
    matrix[1][0] = -3;
    matrix[1][1] = -1;
    matrix[1][2] = 2;
    matrix[2][0] = -2;
    matrix[2][1] = 1;
    matrix[2][2] = 2;

    B[0] = 8;
    B[1] = -11;
    B[2] = -3;

    swap[0] = 0;
    swap[1] = 1;
    swap[2] = 2;
}
```

And the results are proved to be correct in both methods.