ECE/CS 566 Parallel Processing

Programming Assignment III

By: Akhil Kateja (UIN: 661304964)

Darshil Hirpara (UIN: 662079549)

Formulations:

I. Four implemented formulations

In order to calculate the determinant of the kth power of the matrix, I implemented 4 formulations. Serial formulation, Strassen's formulation, Cannon's formulation, DNS formulation

II. Steps to calculate determinant

The algorithm for finding the determinant of the k-th power of the matrix is as follows:

- ➔ Initially generate the array at the logical root process
- ➔ Use cannon or dns formulation distribute the array across the mesh or cube accordingly
- ➔ Calculate the k-th power of the matrix and then Find the determinant

III. Calculating matrix multiplication using associative property

To find the final result we calculated the product of k/2 and square the product. If k is odd, then after squaring we multiply once more with matrix A. To enable associativity in the program, add the -a flag when running.

➔ Conventional Serial Formulation

In this formulation the conventional θ(n3) algorithm for matrix multiplication is implemented. The runtime for this algorithm is θ(n3).

The implementing code is as follows:

```
void blockMatrixMultiplication(float *A, float *B, float *C, int _q) {
    int i, j, k;
    for( i = 0; i < _q; i++ ) {
        for( j = 0; j < _q; j++ ) {
            for( k = 0; k < _q; k++ ) {
                C[i * _q + j] += A[i * _q + k] * B[k * _q + j];
            }
        }
    }
}
```

➔ Strassen Matrix Multiplication

The Strassen algorithm for matrix multiplication faster than serial matrix multiplication algorithm. The running time of Strassen algorithm is $O(N^{2.8074})$

The method implementing the formulation: void performStrassenOp(float *A, float *B, float *C, int _n)

➔ Cannon Formulation

Algorithm is distributed for a mesh of nodes. For matrix multiplication A and B both

are identical so calculation the power of matrix is used by memcpy. A get copied into B. Afterwards A and B are partitioned into p square blocks amongst the processes. In the first communication step, the algorithm aligns the blocks of A and B in such a way that each process multiplies its local sub matrices. To align the matrices, Ai,j is shifted to the left by i steps while all Bi,j are all shifted up by j steps. After the alignment, each block calculates its own local product and local matrix ABi,j gets shifted to the left while local matrix Bi,j gets shifted up. A sequence of sqrt(p) are performed.

The method implementing this formulation: float *performCannonOp(MPI_Comm *comm_new, float *A, float *B, int local_rank, int num_procs)

➔ DNS Formulation

Cannon algorithm uses block 2-D partitioning of the input and output matrices and uses up to n2 processes for nxn matrices, the DNS algorithm can use up to n3 processes in time Θ(logn) by using Ω(p3/2) processes. In the DNS algorithm the processes are arranged in a three-dimensional n x n x n logical array.

In DNS when aligning, we need to initially scatter in the plane k = 0, then perform a one-to-all broadcast of the row (i,k) along dimension j and finally a one-to-all broadcast of the row (i,j) along the k dimension. Also when performing the reduction, each process needs to reduce along the column of the kith plane. Instead of manually calculating the neighbors, synchronizing and performing manually send and receive operations, we created sub-topologies for each of the sub-groups of processors:

```
 // Create a 2-D sub-topology for each of the k-ith dimension
keep_dims[0] = 1; // i
keep_dims[1] = 1; // j
keep_dims[2] = 0; // k
MPI_Cart_sub(*comm_new, keep_dims, &comm_hor);

// Create one along the j dimension
keep_dims[0] = 1;
keep_dims[1] = 0;
keep_dims[2] = 1;
MPI_Cart_sub(*comm_new, keep_dims, &comm_vert);

keep_dims[0] = 0;
keep_dims[1] = 0;
keep_dims[2] = 1;
MPI_Cart_sub(*comm_new, keep_dims, &comm_vert_col);

keep_dims[0] = 0;
keep_dims[1] = 1;
```

```
keep_dims[2] = 0;
MPI_Cart_sub(*comm_new, keep_dims, &comm_hor_row);

keep_dims[0] = 1;
keep_dims[1] = 0;
keep_dims[2] = 0;
MPI_Cart_sub(*comm_new, keep_dims, &comm_hor_col);
```

Like in Cannon's algorithm, an initial alignment needs to be done:

```
MPI_Bcast(A, n_local * n_local, MPI_FLOAT, 0, comm_vert_col);
MPI_Bcast(A, n_local * n_local, MPI_FLOAT, local_coords[2], comm_hor_row);
MPI_Bcast(B, n_local * n_local, MPI_FLOAT, 0, comm_vert_col);
MPI_Bcast(B, n_local * n_local, MPI_FLOAT, local_coords[2], comm_hor_col);
```

After the alignment, each processor calculates it's local product and the result is reduced at the processes of k=0 plane. The code that multiplies locally and performs the reduction operation follows:

```
void performDnsOp(MPI_Comm *comm_new, float *A, float *B, float *C, int local_rank, int num_procs) {
    int i;
    float *AB;
    double start, end, dt;
    AB = (float *) malloc(sizeof(float) * n_local * n_local);
    bzero(AB, sizeof(float) * n_local * n_local);

    blockMatrixMultiplication(A, B, AB, n_local);

    MPI_Reduce(AB, C, n_local * n_local, MPI_FLOAT, MPI_SUM, 0, comm_vert_col);

    free(AB);
 }
```

IV. Obtaining the arguments

The function I use to get the arguments such as n, p, k, etc. is parse_arguments(int argc, char **argv). I get the command line and obtain the string which includes the key information of some information such as the number of nodes and logical processor in each node and the topology I use. The main code to implements the function is following:

```
int parse(int argc, char **argv) {
int i, c;
int option_index = 0;
static struct option long_options[] =
{
```

```
        {"input_method1", required_argument,    0, 'q'},
        {"input_method2", required_argument,    0, 'w'},
        {0, 0, 0, 0}
    };

    char *result = NULL;
    char delims[] = "m";
      while((c = getopt_long (argc, argv, "n:t:q:w:k:ca", long_options,
&option_index)) != -1 ) {
            switch(c) {
           case 'q':
              result = strtok(optarg, delims);
              one_prob = atof(result);
              result = strtok(NULL, delims);
              minus_one_prob = atof(result);
              break;
           case 'w':
              input_method2 = 1;
              result = strtok(optarg, delims);
              input_numbers[0] = atoi(result);
              for( i = 1; i < 4; i++ ) {
                 result = strtok(NULL, delims);
                 input_numbers[i] = atoi(result);
              }
              break;
                  case 'n':
                        n = atoi(optarg);
                        break;
           case 'k':
              k = atoi(optarg);
              break;
           case 'c':
              computerStats = 1;
              break;
           case 'a':
              assoc = 1;
              break;
           case 't':
              if( strcmp(optarg, "serial" ) == 0 ) type = serial;
              else if( strcmp(optarg, "cannon" ) == 0 ) type = cannon;
              else if( strcmp(optarg, "strassen" ) == 0 ) type = strassen;
              else if( strcmp(optarg, "dns" ) == 0 ) type = dns;
              else {
                 fprintf( stderr, "Option -%c %s in incorrect. Allowed values are: serial,
```

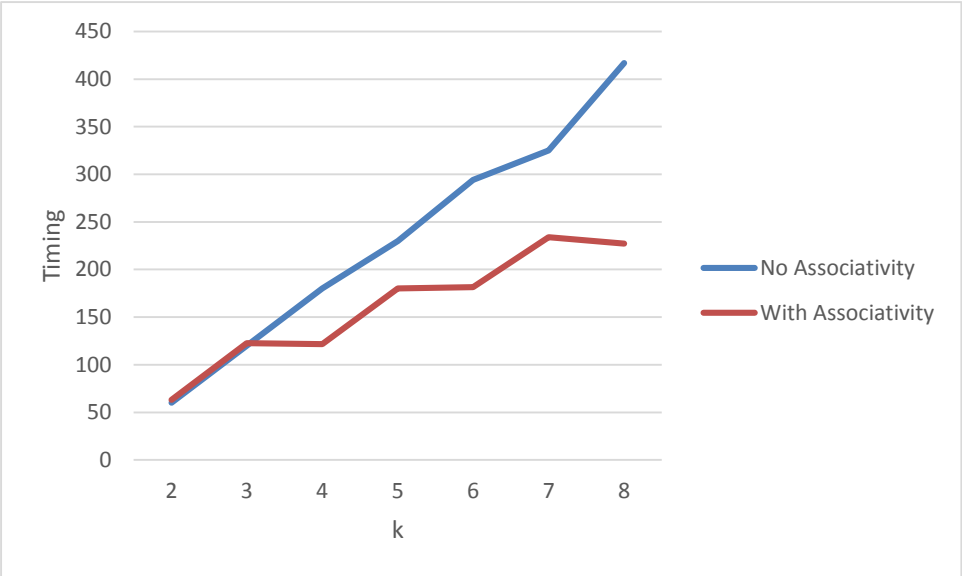
```
strassen, cannon, dns\n", optopt, optarg);
                return 1;
            }
            break;
        case '?':
                if( optopt == 'n' )
                    fprintf (stderr, "Option -%c requires an argument.\n", optopt);
                else if (isprint (optopt))
                    fprintf (stderr, "Unknown option `-%c'.\n", optopt);
                else
                    fprintf (stderr, "Unknown option character `\\x%x'.\n", optopt);
                return 1;
             default:
                fprintf(stderr, "Usage: %s -n <number of numbers> \n", argv[0]);
                fprintf(stderr, "\tExample: %s -n 1000\n", argv[0]);
                return 1;
         }
    }
    return 0;
}
```

Results and Graphs:

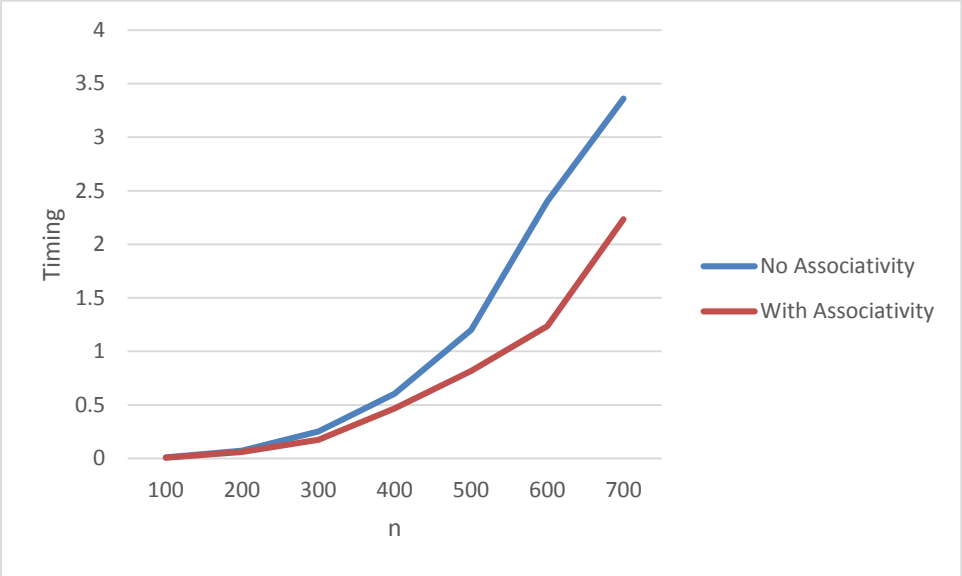
1. The traditional serial method

n	k	p	Timing (in seconds) - no Associativity	Timing (in seconds) - with Associativity
2520	2	1	60	63
2520	3	1	120	122.75
2520	4	1	180	121.62
2520	5	1	230	180.3
2520	6	1	294	181.57
2520	7	1	325	234.12
2520	8	1	417	227.22
100	4	1	0.00887	0.006466
200	4	1	0.07222	0.061045
300	4	1	0.251261	0.174579
400	4	1	0.604852	0.46786
500	4	1	1.2	0.817133
600	4	1	2.4	1.235945
700	4	1	3.362	2.23488

For n=2520



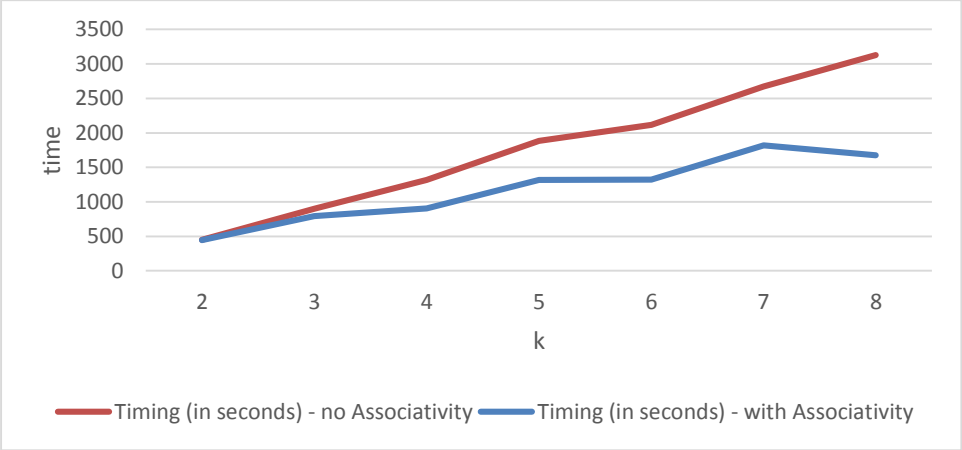
For k=4



2. Strassen Formulation

n	k	p	Timing (in seconds) - no Associativity	Timing (in seconds) - with Associativity
2520	2	1	450.01	443.16
2520	3	1	900.22	794.24
2520	4	1	1316.17	902.22
2520	5	1	1885.13	1315.6
2520	6	1	2116.22	1322.32
2520	7	1	2675.01	1818.92
2520	8	1	3126.99	1676.99

For n=2520

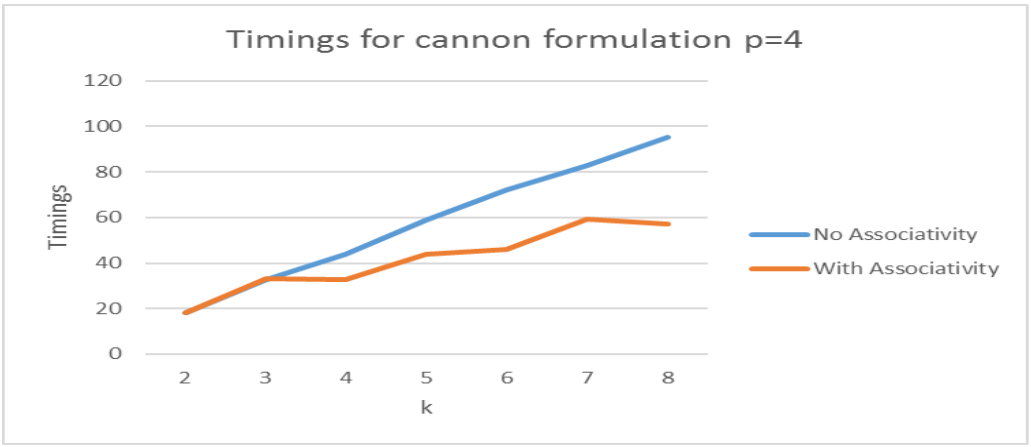


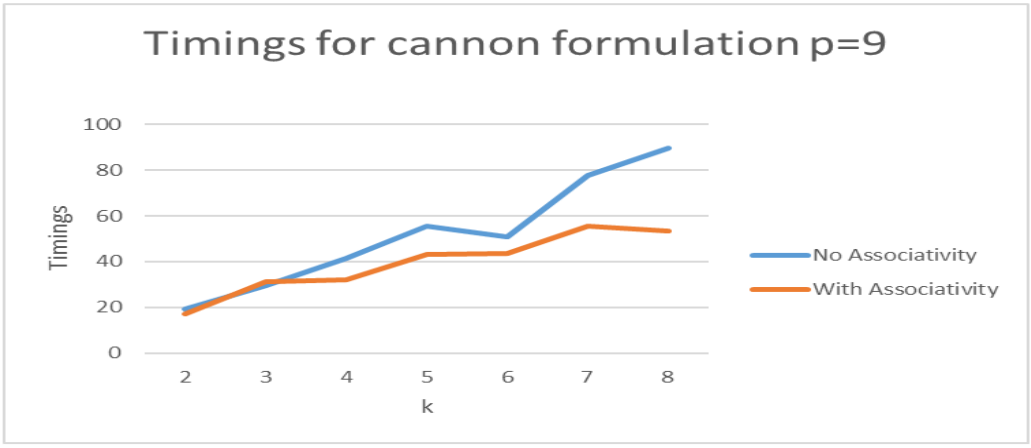
3. Cannon formulation

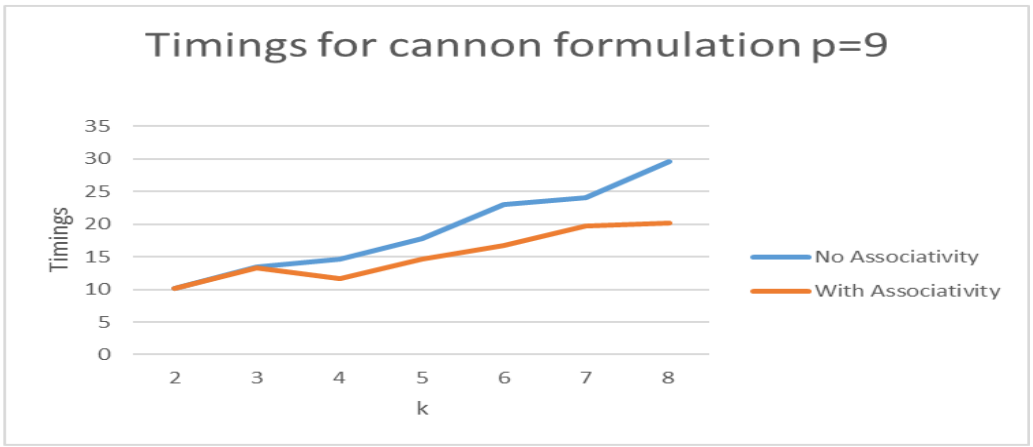
Performance Analysis:
The alignment of the two matrices involves a row wise and column wise circular shift with maximum distance of shift: $\sqrt{p}-1$ and total time: $2(t_s + t_w n^2/p)$. Thus total communication time for alignment is: $T_{comm} = 2(t_s + t_w n^2/p)\sqrt{p}$ Each process performs $\sqrt{p}$ multiplications of $(n/\sqrt{p}) \times (n/\sqrt{p})$ so the total time that each process spends in computation is n^3/p. Thus: $T_p = \frac{n^3}{p} + 2\sqrt{p}t_s + 2t_w\frac{n^2}{\sqrt{p}}$. The isoefficiency function is: $\Theta(p^{3/2})$

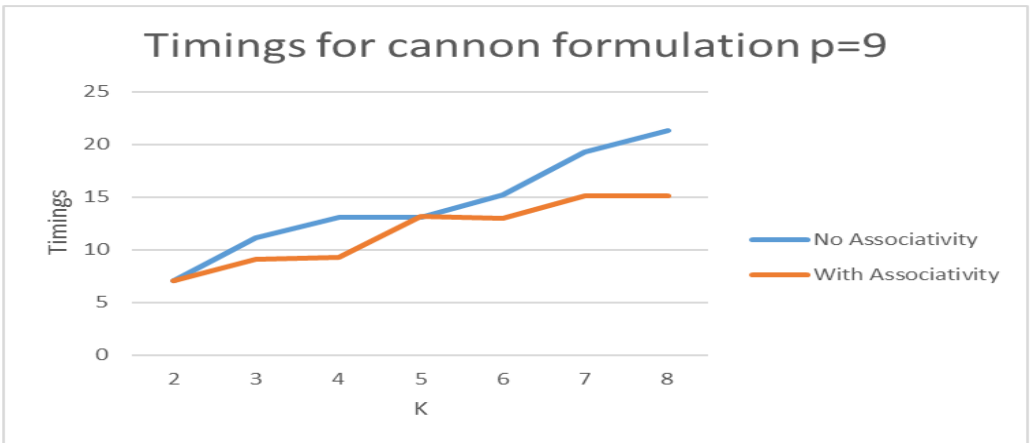
n	k	p	Timing (in seconds) - no Associativity	Timing (in seconds) - with Associativity
2520	2	4	18.02	18.04
2520	3	4	32.89	33.04
2520	4	4	43.9	32.93
2520	5	4	58.82	43.9
2520	6	4	72.04	45.9
2520	7	4	82.82	59.34

2520	8	4	95.37	56.98
2520	2	9	19.22	17.14
2520	3	9	29.39	31.24
2520	4	9	41.53	32.22
2520	5	9	55.61	43.2
2520	6	9	50.64	43.61
2520	7	9	77.71	55.35
2520	8	9	89.71	53.38
2520	2	16	10.21	10.18
2520	3	16	13.37	13.33
2520	4	16	14.7	11.63
2520	5	16	17.75	14.56
2520	6	16	22.97	16.67
2520	7	16	23.99	19.75
2520	8	16	29.59	20.1
2520	2	25	7.08	7.04
2520	3	25	11.14	9.1
2520	4	25	13.09	9.24
2520	5	25	13.06	13.15
2520	6	25	15.25	13.01
2520	7	25	19.32	15.07
2520	8	25	21.36	15.14


Timings for cannon formulation p=4
120
100
80
60
40
20
0
Timings
2
3
4
5
6
7
8
k
No Associativity
With Associativity


Timings for cannon formulation p=9
100
80
60
40
20
0
Timings
2
3
4
5
6
7
8
k
No Associativity
With Associativity


Timings for cannon formulation p=9
35
30
25
20
15
10
5
0
Timings
2
3
4
5
6
7
8
k
No Associativity
With Associativity

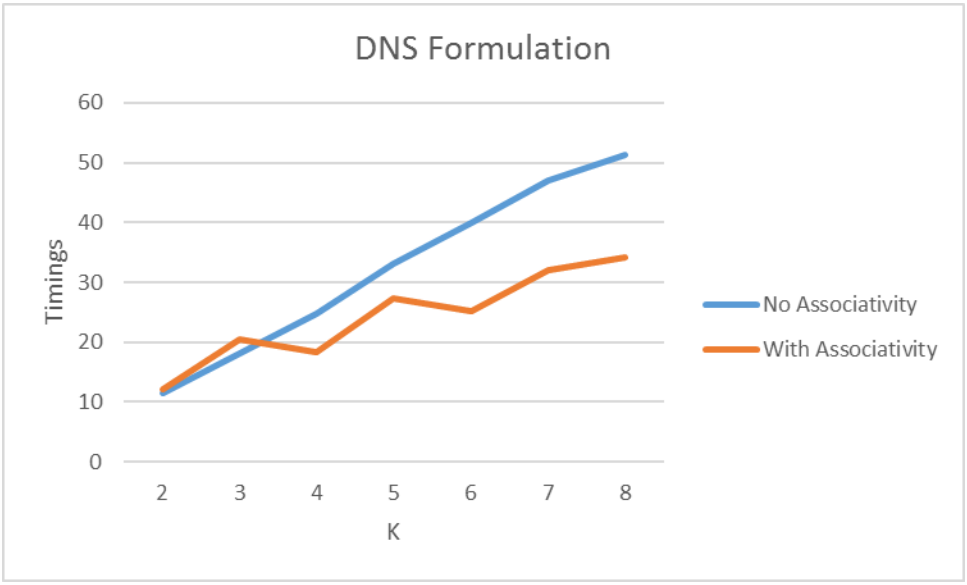

Timings for cannon formulation p=9
25
20
15
10
5
0
Timings
2
3
4
5
6
7
8
K
No Associativity
With Associativity

4. DNS formulation

Performance Analysis

Similarly like in Cannon formulation, by calculating the parallel runtime of DNS, we get: $T_p = \frac{n^3}{p} + t_s \log p + t_w \frac{n^2}{p^{2/3}} \log p$ and isoefficiency function: $\Theta(p(\log p)^3)$. The algorithm is cost-optimal for $n^3 = \Omega(p(\log p)^3)$ or $p = O(n^3/(\log n)^3)$.

n	k	p	Timing (in seconds) - no Associativity	Timing (in seconds) - with Associativity
2520	2	1	11.51	12.09
2520	3	1	18.04	20.43
2520	4	1	24.67	18.3
2520	5	1	33.14	27.27
2520	6	1	39.95	25.11
2520	7	1	47.12	31.93
2520	8	1	51.27	34.1



Conclusion:

1. Comparing the serial implementations

From the charts the Strassen algorithm performs significantly slower than the conventional serial algorithm, even though it is asymptotically slightly faster. It may be due to because of the recursion that the algorithm executes. Each time the program performs recursion, the program counter goes into the stack together with its context registers. Fetching back and forth the context registers is expensive, provided that the matrix is big and does not fit into the cache.

1. The cluster computer is really powerful, since it can help us get the result from the large scaled task faster.
2. The disadvantage of the cluster is, in our experiment data, the generating times are almost the same.