Module:	Introduction to Parallel Programming Techniques
Module ID:	EE4107
Student Name:	Umid Narziev
Student ID:	200234832
Assignment Number:	7
Academic Year:	2020 - 2021

Contents

System Specification

Below, shown the system which was used to run and get the result of the simulation:

CPU:	Intel i5-7200U
Architecture:	Kaby Lake
Segment:	Mobile Processors
The number of cores:	2
Number of threads	4
Clock Frequency	2.50-3.10GHz (Turbo Boost)
Cache levels:	3
Cache level 1 size:	128KBytes
Cache level 2 size:	512Kbytes
Cache level 3 size:	3MBytes
RAM	12 GB
SSD:	250 GB
Operating System:	Ubuntu 20.04.2 LTS
Compiler:	Gcc and its libraries
IDE:	Clion (2020.03)

TASK 1

Code:

```
/* File:
 *     7.1.c
 *
 * Purpose:
 *     Computes a parallel matrix-vector product.  Matrix
 *     is distributed by block rows.  Vectors are distributed by
 *     blocks.  This version uses a random number generator to
 *     generate A and x.  There is some optimization.
 *
 * Compile:
 *    gcc -g -Wall -fopenmp -o 7.1
 *         7.1.c
 * Run:
 *    ./7.1 <thread_count> <m> <n>
 *
 * Input:
 *     None unless compiled with DEBUG flag.
 *     With DEBUG flag, A, x
 *
 * Output:
 *     y: the product vector
 *     Elapsed time for the computation
 *
 * Notes:
 *     1.  Storage for A, x, y is dynamically allocated.
 *     2.  Number of threads (thread_count) should evenly divide both
 *         m and n.  The program doesn't check for this.
 *     3.  We use a 1-dimensional array for A and compute subscripts
 *         using the formula A[i][j] = A[i*n + j]
 *     4.  Distribution of A, x, and y is logical:  all three are
 *         globally shared.
 *     5.  DEBUG compile flag will prompt for input of A, x, and
 *         print y
 */

#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include "timer.h"

#define B_part
/*
 *      define A_part -> for task 7.a
 *      define B_part -> for task 7.b
 */

/* Serial functions */
void Get_args(int argc, char* argv[], int* thread_count_p,
          int* m_p, int* n_p);
void Usage(char* prog_name);
void Gen_matrix(double A[], int m, int n);
void Read_matrix(char* prompt, double A[], int m, int n);
void Gen_vector(double x[], int n);
void Read_vector(char* prompt, double x[], int n);
void Print_matrix(char* title, double A[], int m, int n);
void Print_vector(char* title, double y[], double m);

/* Parallel function */
void Omp_mat_vect(double A[], double x[], double y[],
```

```
              int m, int n, int thread_count);

/*------------------------------------------------------------------*/
int main(int argc, char* argv[]) {
   int     thread_count;
   int     m, n;
   double* A;
   double* x;
   double* y;

   Get_args(argc, argv, &thread_count, &m, &n);

   A = malloc(m*n*sizeof(double));
   x = malloc(n*sizeof(double));
#ifdef A_part
   y = malloc(((8*thread_count)+m)*sizeof(double));
#endif

#ifdef B_part
   y = malloc((m)*sizeof(double));
#endif

#  ifdef DEBUG
  Read_matrix("Enter the matrix", A, m, n);
  Print_matrix("We read", A, m, n);
  Read_vector("Enter the vector", x, n);
  Print_vector("We read", x, n);
#  else
   Gen_matrix(A, m, n);
/* Print_matrix("We generated", A, m, n); */
   Gen_vector(x, n);
/* Print_vector("We generated", x, n); */
#  endif
   double start, finish, elapsed;
   GET_TIME(start);
   Omp_mat_vect(A, x, y, m, n, thread_count);
   GET_TIME(finish);
   elapsed = finish - start;
   printf("Elapsed time = %e seconds\n", elapsed);
#  ifdef DEBUG
   Print_vector("The product is", y, m);
#  else
/* Print_vector("The product is", y, m); */
#  endif

   free(A);
   free(x);
   free(y);

   return 0;
}  /* main */


/*------------------------------------------------------------------
 * Function:  Get_args
 * Purpose:   Get command line args
 * In args:   argc, argv
 * Out args:  thread_count_p, m_p, n_p
 */
void Get_args(int argc, char* argv[], int* thread_count_p,
           int* m_p, int* n_p)  {
```

```c
   if (argc != 4) Usage(argv[0]);
   *thread_count_p = strtol(argv[1], NULL, 10);
   *m_p = strtol(argv[2], NULL, 10);
   *n_p = strtol(argv[3], NULL, 10);
   if (*thread_count_p <= 0 || *m_p <= 0 || *n_p <= 0) Usage(argv[0]);

}  /* Get_args */

/*------------------------------------------------------------------
 * Function:  Usage
 * Purpose:   print a message showing what the command line should
 *            be, and terminate
 * In arg :   prog_name
 */
void Usage (char* prog_name) {
   fprintf(stderr, "usage: %s <thread_count> <m> <n>\n", prog_name);
   exit(0);
}  /* Usage */

/*------------------------------------------------------------------
 * Function:    Read_matrix
 * Purpose:     Read in the matrix
 * In args:     prompt, m, n
 * Out arg:     A
 */
void Read_matrix(char* prompt, double A[], int m, int n) {
   int             i, j;

   printf("%s\n", prompt);
   for (i = 0; i < m; i++)
      for (j = 0; j < n; j++)
         scanf("%lf", &A[i*n+j]);
}  /* Read_matrix */

/*------------------------------------------------------------------
 * Function: Gen_matrix
 * Purpose:  Use the random number generator random to generate
 *    the entries in A
 * In args:  m, n
 * Out arg:  A
 */
void Gen_matrix(double A[], int m, int n) {
   int i, j;
   for (i = 0; i < m; i++)
      for (j = 0; j < n; j++)
         A[i*n+j] = random()/((double) RAND_MAX);
}  /* Gen_matrix */

/*------------------------------------------------------------------
 * Function: Gen_vector
 * Purpose:  Use the random number generator random to generate
 *    the entries in x
 * In arg:   n
 * Out arg:  A
 */
void Gen_vector(double x[], int n) {
   int i;
   for (i = 0; i < n; i++)
      x[i] = random()/((double) RAND_MAX);
}  /* Gen_vector */

/*------------------------------------------------------------------
```

```c
 * Function:       Read_vector
 * Purpose:        Read in the vector x
 * In arg:         prompt, n
 * Out arg:        x
 */
void Read_vector(char* prompt, double x[], int n) {
   int   i;

   printf("%s\n", prompt);
   for (i = 0; i < n; i++)
      scanf("%lf", &x[i]);
}  /* Read_vector */


/*------------------------------------------------------------------
 * Function:  Omp_mat_vect
 * Purpose:   Multiply an mxn matrix by an nx1 column vector
 * In args:   A, x, m, n, thread_count
 * Out arg:   y
 */
void Omp_mat_vect(double A[], double x[], double y[],
              int m, int n, int thread_count) {
   int i, j;
   double temp;

#  pragma omp parallel for num_threads(thread_count) default(none) private(i, j, temp)  shared(A, x, y, m, n)
   for (i = 0; i < m; i++) {
      int my_rank = omp_get_thread_num();
#ifdef A_part
      y[i+(my_rank*8)] = 0.0;
#endif
#ifdef B_Part
      y[i] = 0.0;
      temp = 0.0;
#endif
      for (j = 0; j < n; j++) {
#ifdef A_part
         y[i+(my_rank*8)] += A[i*n+j]*x[j];;
#endif
#ifdef B_part
         temp += A[i*n+j]*x[j];
#endif
      }
#ifdef B_part
      y[i] = temp;
#endif
   }




}  /* Omp_mat_vect */


/*------------------------------------------------------------------
 * Function:    Print_matrix
 * Purpose:     Print the matrix
 * In args:     title, A, m, n
 */
void Print_matrix( char* title, double A[], int m, int n) {
   int   i, j;
```

```
   printf("%s\n", title);
   for (i = 0; i < m; i++) {
      for (j = 0; j < n; j++)
         printf("%4.1f ", A[i*n + j]);
      printf("\n");
   }
}  /* Print_matrix */


/*-------------------------------------------------------------------
 * Function:    Print_vector
 * Purpose:     Print a vector
 * In args:     title, y, m
 */
void Print_vector(char* title, double y[], double m) {
   int   i;

   printf("%s\n", title);
   for (i = 0; i < m; i++)
      printf("%4.1f ", y[i]);
   printf("\n");
}  /* Print_vector */
```

Result:

A) Padded

```
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 7/7.1$ ./7.a 4 8 8000000
Elapsed time = 1.205881e-01 seconds
```

B) With private storage

```
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 7/7.1$ ./7.a 4 8 8000000
Elapsed time = 9.817314e-02 seconds
```

C) Original program

```
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 7/7.1$ ./7.1 4 8 8000000
Elapsed time = 1.985300e-01 seconds
```

Conclusion:

- In fact, the original program has the worst performance among the three.
- The padded version of the program did not show the best result, however, solid improvement is seen, whereas the private variable program has shown a huge improvement.

TASK 2

Code:

```c
/*
 * To compile ->  gcc -g -Wall -fopenmp -o 7.2 7.2.c
 * To run     -> ./7.2 thread_number number_of_element
 */
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <stdlib.h>
#include <time.h>
#include "omp.h"

//#define PRINT // To see the elements

void count_Sort_Serial(int a[], int n);
void count_Sort_Parallel(int a[], int n);
void print(int a[], int n);
void usage(char* prog_name);
void array_Generating(int a[], int n);
int comparator(const void * p1, const void * p2);
int thread_count;

int main(int argc, char* argv[]) {


   int n; /// number of elements
   int *a, *b, *c; /// pointer to the array
   double start, end, temp_Parallel, temp_Serial, temp_qSort;  /// timing variables

   if (argc != 3) usage(argv[0]);
   thread_count = strtol(argv[1], NULL, 10);
   n = strtol(argv[2], NULL, 10);
   a = malloc(n * sizeof(int));
   b = malloc(n * sizeof(int));
   c = malloc(n * sizeof(int));

   array_Generating(a, n);
   memcpy(b, a, n * sizeof(int));// Duplicating the values
   memcpy(c, a, n * sizeof(int));// Duplicating the values
#ifdef PRINT
   print(a, n);
#endif
   //-----------------------------Parallel Count Sort---------------------------------------------------------
   start = omp_get_wtime();
   count_Sort_Parallel(a,n);
   end = omp_get_wtime();
#ifdef PRINT
   printf("After Parallel Count Sort function: \n");
   print(a, n);
#endif
   temp_Parallel = end - start;
   //-----------------------------Serial Count Sort---------------------------------------------------------
   start = omp_get_wtime();
   count_Sort_Serial(b,n);
   end = omp_get_wtime();
#ifdef PRINT
   printf("After Serial Count Sort function: \n");
   print(b, n);
#endif
   temp_Serial = end - start;
```

```c
    //------------------------------qSort Count Sort------------------------------------------------------------
    start = omp_get_wtime();
    qsort(c, n, sizeof(int), comparator);
    end = omp_get_wtime();
#ifdef PRINT
    printf("After qSort Count Sort function: \n");
    print(b, n);
#endif
    temp_qSort = end - start;
    //-----------------------------The result printing -------------------------------------------------------
    printf("The elapsed time is %e for Parallel count sort function with %d elements.\n", temp_Parallel, n);
    printf("The elapsed time is %e for Serial count sort function with %d elements.\n", temp_Serial, n);
    printf("The elapsed time is %e for Qsort count sort function with %d elements.\n", temp_qSort, n);
    return 0;
}

void usage(char* prog_name) {

    fprintf(stderr, "usage: %s <number of threads>, <number of elements>\n", prog_name);
    exit(0);
}
void array_Generating(int a[], int n){

    for (int i = 0; i < n; ++i) {
        srand(i);
        a[i] = rand() % n;
    }
}
void print(int a[], int n){
    for (int i = 0; i < n; i++) {
        printf("%d\t", a[i]);
    }
    printf("\n");
}
void count_Sort_Serial(int a[], int n){
    int count;
    int *temp = malloc(n*sizeof(int));

    for (int i = 0; i < n; ++i) {
        count = 0;
        for (int j = 0; j < n; ++j) {
            if(a[j]<a[i]){
                count++;
            } else if((a[j] == a[i]) && (j < i)){
                count++;
            }
        }
        temp[count] = a[i];
    }

    memcpy(a, temp, n * sizeof(int));
    free(temp);
}
void count_Sort_Parallel(int a[], int n){
    int count, i, j;
    int *temp = malloc(n*sizeof(int));
#pragma omp parallel for num_threads(thread_count) default(none) shared(a, n,  temp) private(j, i, count)
    for (i = 0; i < n; ++i) {
        count = 0;
        for (j = 0; j < n; ++j) {
            if(a[j] < a[i]){
                count++;
```

```
        } else if((a[j] == a[i]) && (j < i)){
            count++;
        }
      }
#pragma omp critical
      {
        temp[count] = a[i];
      };

   }

   memcpy(a, temp, n * sizeof(int));

   free(temp);
}
int comparator (const void * p1, const void * p2){
   return (*(int*)p1 - *(int*)p2);
}
```

Result:

```
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 7/7.2$ gcc -g -Wall -fopenmp -o 7.2 7.2.c
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 7/7.2$ ./7.2 4 1000
The elapsed time is 1.217650e-02 for Parallel count sort function with 1000 elements.
The elapsed time is 1.302248e-02 for Serial count sort function with 1000 elements.
The elapsed time is 1.743030e-04 for Qsort count sort function with 1000 elements.
```

Conclusion:

- The private variables are array j, i, count, shared variables are array a, n, temp.
- No, there is no loop cared dependency as i-private.
- The memcpy function is not IO-bound, not CPU-bound, so parallelizing will not give much performance boost. And also, parallelizing it not safe as we don’t know how it is implemented, memory location may be overwritten.
- The qSort function has the highest performance among all, the parallelising of count sort has given benefit but not as much as qsort.

TASK 3

Code:

```
/*
 * To compile -> gcc -g -Wall -fopenmp -o 7.3 7.3.c
 * To run     -> ./7.3
 * export OMP_SCHEDULE="guided, 500"
 */


#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include "omp.h"

void row_Oriented_Inner(double A[], double b[], double x[], int n, int thread_count);
void column_Oriented_Inner(double A[], double b[], double x[], int n, int thread_count);
void generatorMatrixVector(double A[], double b[],  int n);
void matrixVectorPrint(double A[], double b[],  int n);
void printVector(double x[], int n);
int main() {
   int n = 10000;
   double *a = malloc(n * n * sizeof(double ));
   double *b = malloc(n * n * sizeof(double ));
   double *x =  malloc(n * sizeof(double ));
   double *y =  malloc(n * sizeof(double ));
   double start, end;
   generatorMatrixVector(a, b, n);
   for (int thread_count  = 1; thread_count  < 16; thread_count<<=1) {
      start = omp_get_wtime();
      column_Oriented_Inner(a, b, x, n, thread_count);
      end = omp_get_wtime();
      printf("[Column] The elapsed time -> %f, thread_count -> %d, n -> %d, thread_count->%d\n", end - start,
           thread_count, n, thread_count);
   }
   for (int thread_count  = 1; thread_count  < 16; thread_count<<=1) {
      start = omp_get_wtime();
      row_Oriented_Inner(a, b, y, n, thread_count);
      end = omp_get_wtime();
      printf("[Row] The elapsed time -> %f, thread_count -> %d, n -> %d, thread_count->%d\n", end - start,
           thread_count, n, thread_count);
   }
   return 0;
}

void row_Oriented_Inner(double A[], double b[], double x[], int n, int thread_count){
   double temp;
   int row, col;
#pragma omp parallel default(none) num_threads(thread_count) \
   shared(n, b, A, temp, x) private(col, row)
   for ( row = n-1; row >= 0; row--) {
#pragma omp single
      temp = b[row];
#pragma omp for reduction(+:temp) schedule(runtime)
      for ( col = row + 1; col < n; col++) {
         temp += -A[row*n+col] * x[col];
      }
#pragma omp single  // using ciritical gives strange result as it is distributed over process so I used again
single
```

```c
      x[row] = temp /  A[row*n+row] ;
   }
}

void column_Oriented_Inner(double A[], double b[], double x[], int n, int thread_count){
   int col, row;
#pragma omp parallel default(none) num_threads(thread_count) \
   shared(n, b, A, x) private(col, row)
   {
#pragma omp for
      for (int row = 0; row < n; ++row) {
         x[row] = b[row];
      }
      for (col = n - 1; col >= 0 ; col--) {
#pragma omp single
         x[col] /= A[col*n+col];
#pragma omp for schedule(runtime)
         for (row = 0; row < col; row++) {
            x[row] +=- A[row*n+col] * x[col];
         }
      }
   }
}

void generatorMatrixVector(double A[], double b[],   int n){
   srand(1);
   for (int col = 0; col < n; ++col) {
      for (int row = 0; row < n; ++row) {
         A[row*n+col] =  random()/((double) RAND_MAX);
      }
   }
   srand(2);
   for (int i = 0; i < n; ++i) {
      b[i] =  random()/((double) RAND_MAX);
   }
}
void matrixVectorPrint(double A[], double b[],   int n){

   printf("The generated matrix\n");
   for (int col = 0; col < n; ++col) {
      for (int row = 0; row < n; ++row) {
         printf("%f\t", A[row*n+col]);
      }
      printf("\n");
   }
   printf("The generated vector\n");
   for (int i = 0; i < n; ++i) {
      printf("%f\t", b[i]);
   }
   printf("\n");

}
void printVector(double x[], int n){

   printf("The vector is: \n");
   for (int i = 0; i < n; ++i) {
      printf("%10.4f\t", x[i]);
   }
   printf("\n");

}
```

Result:

1. Default Scheduler:

```
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 7/7.3$ gcc -g -Wall -fopenmp -o 7.3 7.3.c
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 7/7.3$ ./7.3
[Column] The elapsed time -> 2.805761, thread_count -> 1, n -> 10000, thread_count->1
[Column] The elapsed time -> 2.751760, thread_count -> 2, n -> 10000, thread_count->2
[Column] The elapsed time -> 1.678471, thread_count -> 4, n -> 10000, thread_count->4
[Column] The elapsed time -> 1.801062, thread_count -> 8, n -> 10000, thread_count->8
[Row] The elapsed time -> 0.900965, thread_count -> 1, n -> 10000, thread_count->1
[Row] The elapsed time -> 1.440938, thread_count -> 2, n -> 10000, thread_count->2
[Row] The elapsed time -> 0.982260, thread_count -> 4, n -> 10000, thread_count->4
[Row] The elapsed time -> 1.605797, thread_count -> 8, n -> 10000, thread_count->8
```

2. Static, chunk size = 8:

```
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 7/7.3$ export OMP_SCHEDULE="static, 8"
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 7/7.3$ ./7.3
[Column] The elapsed time -> 0.684000, thread_count -> 1, n -> 10000, thread_count->1
[Column] The elapsed time -> 0.435846, thread_count -> 2, n -> 10000, thread_count->2
[Column] The elapsed time -> 0.384885, thread_count -> 4, n -> 10000, thread_count->4
[Column] The elapsed time -> 0.813419, thread_count -> 8, n -> 10000, thread_count->8
[Row] The elapsed time -> 0.177707, thread_count -> 1, n -> 10000, thread_count->1
[Row] The elapsed time -> 0.172560, thread_count -> 2, n -> 10000, thread_count->2
[Row] The elapsed time -> 0.141851, thread_count -> 4, n -> 10000, thread_count->4
[Row] The elapsed time -> 0.641442, thread_count -> 8, n -> 10000, thread_count->8
```

3. Guided, chunk size = 8:

```
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 7/7.3$ export OMP_SCHEDULE="guided, 8"
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 7/7.3$ ./7.3
[Column] The elapsed time -> 0.672911, thread_count -> 1, n -> 10000, thread_count->1
[Column] The elapsed time -> 0.394495, thread_count -> 2, n -> 10000, thread_count->2
[Column] The elapsed time -> 0.347416, thread_count -> 4, n -> 10000, thread_count->4
[Column] The elapsed time -> 0.632058, thread_count -> 8, n -> 10000, thread_count->8
[Row] The elapsed time -> 0.178246, thread_count -> 1, n -> 10000, thread_count->1
[Row] The elapsed time -> 0.111082, thread_count -> 2, n -> 10000, thread_count->2
[Row] The elapsed time -> 0.162991, thread_count -> 4, n -> 10000, thread_count->4
[Row] The elapsed time -> 0.497952, thread_count -> 8, n -> 10000, thread_count->8
```

4. Dynamic, chunk size = 8:

```
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 7/7.3$ export OMP_SCHEDULE="dynamic, 8"
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 7/7.3$ ./7.3
[Column] The elapsed time -> 0.927137, thread_count -> 1, n -> 10000, thread_count->1
[Column] The elapsed time -> 0.577075, thread_count -> 2, n -> 10000, thread_count->2
[Column] The elapsed time -> 0.576333, thread_count -> 4, n -> 10000, thread_count->4
[Column] The elapsed time -> 0.681261, thread_count -> 8, n -> 10000, thread_count->8
[Row] The elapsed time -> 0.278211, thread_count -> 1, n -> 10000, thread_count->1
[Row] The elapsed time -> 0.230488, thread_count -> 2, n -> 10000, thread_count->2
[Row] The elapsed time -> 0.190849, thread_count -> 4, n -> 10000, thread_count->4
[Row] The elapsed time -> 0.575282, thread_count -> 8, n -> 10000, thread_count->8
```

5. Auto:

```
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 7/7.3$ export OMP_SCHEDULE="auto"
umid@umid-Lenovo-ideapad-320-15IKB:/media/umid/Data/Aston University/Subjects/TP2/EE4107 - Introduction to Parallel Programming Techniques/Assignments/Assignment - 7/7.3$ ./7.3
[Column] The elapsed time -> 0.679225, thread_count -> 1, n -> 10000, thread_count->1
[Column] The elapsed time -> 0.350097, thread_count -> 2, n -> 10000, thread_count->2
[Column] The elapsed time -> 0.330308, thread_count -> 4, n -> 10000, thread_count->4
[Column] The elapsed time -> 0.782548, thread_count -> 8, n -> 10000, thread_count->8
[Row] The elapsed time -> 0.179200, thread_count -> 1, n -> 10000, thread_count->1
[Row] The elapsed time -> 0.102238, thread_count -> 2, n -> 10000, thread_count->2
[Row] The elapsed time -> 0.101052, thread_count -> 4, n -> 10000, thread_count->4
[Row] The elapsed time -> 0.668183, thread_count -> 8, n -> 10000, thread_count->8
```

Conclusion:

- No, outer loop of the row-oriented algorithm can not be parallelized, x[row] = b[row] and x[row]/=A[row][row], introduce loop carried dependency.
- Yes, the inner loop of the row-oriented algorithm can be parallelized.
- No, outer loop of column-oriented algorithm can not be parallelized as x[col]/=A[col][col] introduce loop-dependency.
- Yes, the inner loop of the column-oriented algorithm can be parallelized, and code is written.
- The guided scheduler has performance, with ~0.63 seconds for row-oriented algorithm and ~0.49 seconds for the column-oriented algorithm for 8 threads.