COMP605/CS605: Scientific Computing, Spring 2023
Assignment 3 - Due Friday 3/24/23

Under your home directory create a directory assign3.
Then cd assign3 and create two directories: CountSort and LinearSystem.

1. (30 points) Count sort is a simple serial algorithm that can be implemented as follows:

```
void Count_sort(int a[], int n)
{
    int i, j, count;
    int* temp = malloc(n*sizeof(int));

    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++;

        temp[count] = a[i];
    }

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

    free(temp);
}
```

   The idea is that for each element $a[i]$ in the list a, one counts the number of elements in the list that are less than $a[i]$. Then one inserts $a[i]$ into a temporary list using the subscript determined by the count. If the list contains equal elements, the code increments the count for equal elements on the basis of the subscripts. If both $a[i] == a[j]$ and $j < i$, then one counts $a[j]$ as being "less than" $a[i]$.

   After the algorithm has completed, one overwrites the original array with the temporary array using the string library function *memcpy*.

   (a) If one tries to parallelize the outer loop, which variables should be private and which shared?

   (b) Are there any loop-carried dependencies in the previous parallelization? Explain your answer.

   (c) Is it possible to parallelize the call to *memcpy*? Explain your answer.

   (d) Write an OpenMP program that includes your parallel implementation of *Count_sort*.

   (e) How does the performance of your parallelization of *Count_sort* compared to serial *Count_sort*?

   (f) How does it compare to the serial *qsort* library function?

2. (70 points) When one solves a large linear system $Ax = b$, one often uses Gaussian elimination followed by backward substitution. Gaussian elimination converts an $n \times n$ linear system into an upper triangular linear system by using the "row operations"

   - Add a multiple of one row to another row.
   - Swap two rows.

- Multiply one row by a nonzero constant.

One can devise a couple of serial algorithms for back substitution. The "row-oriented" version is

```
for (row = n-1; row >= 0; row--)
{
    x[row] = b[row];

    for (col = row+1; col < n; col++)
        x[row] -= A[row][col]*x[col];

    x[row] /= A[row][row];
}
```

Here the right-hand side of the system is stored in array b, the two-dimensional array of coefficients is stored in array A, and the solutions are stored in array x. An alternative is the following "column-oriented" algorithm:

```
for (row = 0; row < n; row++)
    x[row] = b[row];

for (col = n-1; col >=0; col--)
{
    x[col] /= A[col][col];

    for (row = 0; row < col; row++)
        x[row] -= A[row][col]*x[col];
}
```

(a) Determine whether the outer loop of the row-oriented algorithm can be parallelized.

(b) Determine whether the inner loop of the row-oriented algorithm can be parallelized.

(c) Determine whether the outer loop of the column-oriented algorithm can be parallelized.

(d) Determine whether the inner loop of the column-oriented algorithm can be parallelized.

(e) Write an OpenMP program for each of the loops that you determine could be parallelized. Consider using *#pragma omp single*.

(f) Modify your parallel loop with a *schedule(runtime)* clause and test the program with various schedules.

(g) If your upper triangular system has 10,000 variables, which schedule gives the best performance? Explain your answer.

(h) Use OpenMP to implement a program that does the Gaussian elimination (assuming that the linear system is invertible and that there is no row-swapping).

To test your algorithm consider matrices of the form

$$A = \begin{pmatrix} T & I & 0 & \cdots & & 0 \\ I & T & I & 0 & \cdots & 0 \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ 0 & \cdots & 0 & I & T & I \\ 0 & \cdots & & 0 & I & T \end{pmatrix}, \qquad T = \begin{pmatrix} -4 & 1 & 0 & \cdots & & 0 \\ 1 & -4 & 1 & 0 & \cdots & 0 \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ 0 & \cdots & 0 & 1 & -4 & 1 \\ 0 & \cdots & & 0 & 1 & -4 \end{pmatrix},$$

where I is the identity matrix that has the same size of the square matrix T.