COL380 - Parallel and Distributed Programming : Assignment - 2

Harish Kumar Yadav - 2018CS10337
Param Khakhar - 2018CS10362

05 April 2021

Introduction

The following is a writeup for the Assignment-2, Parallel and Distributed Programming (COL380). There are sections corresponding to each of the four strategies implemented, which also contains the answers to the questions.

Changes to the Base Code

The code provided in the assignment specification was modified in order to achieve better results after using OpenMP for parallelism. The two versions are as presented below:

```
void crout(double const **A, double **L, double **U, int n) {

  int i, j, k;
  double sum = 0;

  for (i = 0; i < n; i++) {
    U[i][i] = 1;
  }

  for (j = 0; j < n; j++) {
    for (i = j; i < n; i++) {
```

```
      sum = 0;
      for (k = 0; k < j; k++) {
        sum = sum + L[i][k] * U[k][j];
      }
      L[i][j] = A[i][j] - sum;
    }

    for (i = j; i < n; i++) {
      sum = 0;
      for(k = 0; k < j; k++) {
        sum = sum + L[j][k] * U[k][i];
      }
      if (L[j][j] == 0) {
        exit(0);
      }
      U[j][i] = (A[j][i] - sum) / L[j][j];
    }
  }
}

void crout3(double **S, double const **A, double **L, double **U, int n)
{
      //S is an nxn auxiliary array initialized with 0's.

      int i, j, k;
      for (i = 0; i < n; i++) {
          U[i][i] = 1;
      }

      for(j = 0; j < n; ++j){
          L[j][j] = A[j][j]-S[j][j];
          U[j][j] = (A[j][j]-S[j][j])/L[j][j];
          double val = L[j][j];
          for(i = j+1; i < n; ++i){
              L[j][i] = A[i][j] - S[i][j];
              U[j][i] = (A[j][i] - S[j][i])/val;
          }
```

```
        for(int row1 = j; row1 < n; ++row1){
            for(int row2 = j; row2 < n; ++row2){
                S[row1][row2] += L[j][row1]*U[j][row2];
            }
        }
    }
}
```

Following are the salient features for the modified code:

- The indexing in the array is done in consistence with the **row-major** order in order to have maximum cache hits.
- The overall structure of the code is very simple, thus enabling **compiler optimizations** such as vectorized operations.

Strategy-1 : Using Parallel For

The following statistics were observed for our implementation when run on a diagonal matrix of size 2000. We executed the program for 50 iterations and computed the mean, std, and median time for the executions. We found that the median is a better statistic for the measured time as the outliers misrepresented the mean time as well as standard deviation.

# Threads	Median Time in s	Speedup	Efficiency
1	12.498	1.00	1.00
2	6.842	1.826	0.913
4	4.624	2.70	0.675
8	4.68	2.667	0.333
16	4.92	2.54	0.159

Description:

```
  int i, j, k;

#pragma omp parallel for
for (i = 0; i < n; i++) {
  U[i][i] = 1;
  }

for(j = 0; j < n; ++j){
      #pragma omp parallel
      {
          L[j][j] = A[j][j]-S[j][j];
          U[j][j] = (A[j][j]-S[j][j])/L[j][j];
          double val = L[j][j];
          #pragma omp for
          for(i = j+1; i < n; ++i){
              L[j][i] = A[i][j] - S[i][j];
              U[j][i] = (A[j][i] - S[j][i])/val;
          }
          #pragma omp for
```

```
            for(int row1 = j; row1 < n; ++row1){
                for(int row2 = j; row2 < n; ++row2){
                    S[row1][row2] += L[j][row1]*U[j][row2];
                }
            }
        }
}
```

- In this approach we used #pragma omp for all the double nested loops in the function crout3 as there are no loop carried dependencies.
- The new proposed algorithm crout3, requires the nested for loops to be completed in order and therefore we haven't specified the nowait clause in the pragma omp for.
- Static allocation would be better as the tasks are more or less similar, and using dynamic would also lead to other maintenance overheads.

Correctness:

The outer loop index (j) is the common variable among all the nested for loops. For the first double nested loop with loop variable i, there aren't any loop carried dependencies. However, there are locations which are accessed more than once within the triple nested last for loop. But, the shared locations after every iteration of the outer loop. Using #pragma omp for for ensures completion of all the threads in the current iteration before proceeding for the next iteration of the outermost loop, thereby mitigating the issue of data race.

Strategy-2 : Using Sections

# Threads	Median Time in s	Speedup	Efficiency
1	12.498	1.00	1.00
2	6.665	1.875	0.938
4	5.094	2.453	0.613
8	4.939	2.53	0.316
16	4.675	2.673	0.167

Description:

```
  void sections(double** S, double const **A, double **L, double **U, int n, int t

    for (int i = 0; i < n; i++) {
        U[i][i] = 1;
    }

    for(int j = 0; j < n; ++j){
      L[j][j] = A[j][j]-S[j][j];
          double val = L[j][j];

      #pragma omp parallel sections
      {
            #pragma omp section
            {
                for(int i = j+1; i < n; ++i){
                    L[j][i] = A[i][j] - S[i][j];
                }
            }
            #pragma omp section
            {
                for(int i = j; i < n; ++i){
                    U[j][i] = (A[j][i] - S[j][i])/val;
                }
            }
}
```

```
if(t == 2){
  int fir = (j+n)/2;
  #pragma omp parallel sections
  {
  #pragma omp section
  {
      for(int row1= j; row1 < fir; ++row1){
          for(int row2 = j; row2 < n; ++row2){
              S[row1][row2] += L[j][row1]*U[j][row2];
          }
      }
  }
#pragma omp section
{
      for(int row1= fir; row1 < n; ++row1){
          for(int row2 = j; row2 < n; ++row2){
              S[row1][row2] += L[j][row1]*U[j][row2];
          }
      }
}
}
}
```

.
.
.
.

- In this approach, we split the statements of the first nested for loop in crout3, into two different loops which can run concurrently and thus enabling us to use the sections clause specified by OpenMP.

- For the last triple nested for loop, we again forced parallelism by splitting the iterations into different mutually exclusive and exhaustive ranges. The implementation uses the fact that the number of threads is hard-coded at 2, 4, 8, and 16.

Correctness:

The shared variable among the inner nested loops is (j), since the loops with variables i, execute concurrently, by individual threads at a time, they don't perform write operations to the same address. There are shared memory locations for different iteration of the outer variable (j) of the S array but the completion of sections parallel clause in each loop iteration, prevents multiple writes at the same address.

Strategy-3 : Using Parallel For and Sections

# Threads	Median Time in s	Speedup	Efficiency
1	12.498	1.00	
2	6.515	1.918	0.959
4	5.061	2.469	0.617
8	4.875	2.564	0.32
16	4.147	3.104	0.188

Description:

```
void sections(double** S, double const **A, double **L, double **U, int n, int t
  #pragma omp for
  for (int i = 0; i < n; i++) {
      U[i][i] = 1;
  }

  for(int j = 0; j < n; ++j){
    L[j][j] = A[j][j]-S[j][j];
        double val = L[j][j];

    #pragma omp parallel sections
    {
          #pragma omp section
          {
              for(int i = j+1; i < n; ++i){
                  L[j][i] = A[i][j] - S[i][j];
              }
          }
          #pragma omp section
          {
              for(int i = j; i < n; ++i){
                  U[j][i] = (A[j][i] - S[j][i])/val;
              }
          }
```

```
}
if(t == 2){
  int fir = (j+n)/2;
  #pragma omp parallel sections
  {
  #pragma omp section
  {
      for(int row1= j; row1 < fir; ++row1){
          for(int row2 = j; row2 < n; ++row2){
              S[row1][row2] += L[j][row1]*U[j][row2];
          }
      }
  }
#pragma omp section
{
      for(int row1= fir; row1 < n; ++row1){
          for(int row2 = j; row2 < n; ++row2){
              S[row1][row2] += L[j][row1]*U[j][row2];
          }
      }
}
}
}
    .
    .
    .
```

In this approach, we simply experimented with the different combinations of the above two approaches and thus have implemented the best combination. Strategy 2 gave better results, therefore it was used, however the for loop for initializing the U matrix has been parallelized now using #pragma omp for.

Correctness:

Since, the program for strategy is formed by the combination of strategies 1 and 2, the same explanation holds for the correctness and data races for this part.

Strategy-4 : Distributed Version

# Processes	Median Time in s	Speedup	Efficiency
1	11.153	1.00	1.00
2	18.527	0.602	0.301
4	31.229	0.357	0.09
8	53.528	0.208	0.026

Description:

```
for (i = 0; i < n; i++) {
  U[i][i] = 1;
}

for(j = 0; j < n; ++j){
  L[j][j] = matrix[j][j]-S[j][j];
  U[j][j] = (matrix[j][j]-S[j][j])/L[j][j];
  double val = L[j][j];
  for(i = j+1; i < n; ++i){
    L[j][i] = matrix[i][j] - S[i][j];
    U[j][i] = (matrix[j][i] - S[j][i])/val;
}

  for(int row1 = j; row1 < n; ++row1){
    if(row1 % threads == my_rank)
    {
      for(int row2 = j; row2 < n; ++row2){
        S[row1][row2] += L[j][row1]*U[j][row2];
      }
    }
    MPI_Bcast(S[row1], n, MPI_DOUBLE, row1 % threads, MPI_COMM_WORLD);
  }
}
```

- The objective while designing the code for this part was to have minimal, simpler code with as less send-receive or broadcast operations,

as they are expensive. However, the least number of such operations which need to be performed would at least be $O(n^2)$. Therefore, only the triple nested operations are divided among the processes.

- The modulo operation is used to perform the splitting of tasks, which takes place in constant time.

Correctness:

The correctness of the program is discussed in the previous strategies, however the only difference in this would be the individual processes writing to the 2D array, S. No two processes write to the same location due to the scheme for the division of work adopted in the program. Therefore, the issue of data race doesn't arise.

Remarks:

- For strategies 1, 2, and 3 the performance is better than the sequential case. For strategy-4, the overhead of performing a broadcast is significantly more as compared to the size of problem (2000 size 2D array) and therefore, the value of speedup obtained is < 1.
- For all the strategies, the efficiency decreases on increasing the number of threads/processes which indeed is theoretically consistent.
- The time mentioned in the table includes the time for reading the input and performing the crout matrix matrix decomposition.