COL380: Parallel and Distributed Programming
Assignment-2

Aarunish Sinha Jai Arora

1 Terminology

This is the original serialized version of the Crout Matrix Decomposition. It has a number of loops (all of them are Natural Loops).

```
void crout(double const **A, double **L, double **U, int n) {
        int i, j, k;
        double sum = 0;
        /* Loop #1 */
        for (i = 0; i < n; i++) {
                U[i][i] = 1;
        }
        /* Loop #2 */
        for (j = 0; j < n; j++) {
            /* Loop #3 */
                for (i = j; i < n; i++) {
                        sum = 0;
                        /* Loop #4 */
                        for (k = 0; k < j; k++) {
                                sum = sum + L[i][k] * U[k][j];
                        }
                        L[i][j] = A[i][j] - sum;
                }
                /* Loop #5 */
                for (i = j; i < n; i++) {
                        sum = 0;
                        /* Loop #6 */
                        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];
                }
        }
}
```

Denote Loop #i by L_i. Now it can be observed that:

- L_3 is nested inside L_2
- L_4 is nested inside L_3
- L_5 is nested inside L_2
- L_6 is nested inside L_5

2 Strategy-0:

Matrix Size	Time (in seconds)
32	0.001436
256	0.067976
1024	1.940988
2048	15.604980
4096	202.770537

Table 1: Time Taken

The trend here is pretty obvious: the time taken increases as the problem size increases.

3 Strategy-1:

In this strategy we were supposed to use the `parallel for` construct of `OpenMP`.

3.1 Implementation

We have parallelized three `for` loops in Crout Matrix Decomposition Algorithm:

- The first loop where all the diagonal elements of the upper triangular matrix are set to 1
- The first inner loop where we write the elements in the lower triangular matrix
- and the second inner loop in where we write the remaining elements of the upper triangular matrix.

We have also kept the variables `sum, i, j, k` as private to the threads.

3.2 Handling Data Races

The loop L_1 has no loop dependencies (both loop carried and loop independent) and hence can be directly parallelized.

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

The loop L_2 has L_3 and L_5 nested inside it. We can observe see that the j^{th} iteration of L_2 computes $L[i][j]\ \forall i \in \{0, 1, ..., n-1\}$, and

$$L[i][j] = A[i][j] - \texttt{sum}$$
$$= A[i][j] - \sum_{k=0}^{j-1} L[i][k] * U[k][j]$$

so $L[i][j]$ depends on $L[i][0]$, ... , $L[i][j-1]$, which would have been computed in the previous iterations of L_2, indicating the presence of anti-dependency across loop iterations. So we cannot directly parallelize L_2 by dividing the iterations across threads because it has loop carried dependencies.

Now the loops L_4 and L_6 cannot be parallelized as they have loop carried dependencies. For a given `j`, there is no loop dependency in L_3 as it's i^{th} iteration reads from $L[i][k]\ \forall k \in \{0, ..., j-1\}$ (which have been already computed in previous iterations of L_2) and writes to $L[i][j]$. Same can be said about L_5, so we can directly parallelize both of them as follows:

```
/* Loop #3 */
#pragma omp for
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;
```

```
    }

    /* Loop #5 */
    #pragma omp for
    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];
    }
```

Also, for each iteration of L_2, L_3 needs to complete before L_5 because the former computes $L[j][j]$ and the latter uses $L[j][j]$, hence there is a data dependency in L_2 (Loop independent dependency). This data race is implicitly handled by our constructs as there is an implicit barrier after each #`pragma omp for` construct.

3.3 Analysis

Matrix Size	2 Threads	4 Threads	8 Threads	16 Threads
32	0.002402	0.003458	0.004463	0.007429
256	0.062456	0.058277	0.061176	0.073021
1024	1.394235	1.042803	0.850068	0.877420
2048	10.875080	6.763208	4.787918	4.954131
4096	120.704161	70.056022	46.960543	43.882381

Table 2: Time Taken (in seconds)

Matrix Size	2 Threads	4 Threads	8 Threads	16 Threads
32	0.598	0.415	0.322	0.193
256	1.088	1.166	1.111	0.931
1024	1.392	1.861	2.283	2.212
2048	1.435	2.307	3.259	3.150
4096	1.680	2.894	4.318	4.621

Table 3: Speed-Up

Matrix Size	2 Threads	4 Threads	8 Threads	16 Threads
32	0.299	0.104	0.040	0.012
256	0.544	0.292	0.139	0.058
1024	0.696	0.465	0.285	0.138
2048	0.717	0.577	0.407	0.197
4096	0.840	0.724	0.540	0.288

Table 4: Efficiency

Observations:

- On smaller matrix size, the synchronization overhead is large, so the speedup < 1, and the time taken increases and speedup and efficiency decrease as the number of threads increase.
- On larger input sizes, the total time taken decreases as the number of threads increase. In these cases, the speedup is > 1 and it also increases with the number of threads, but the efficieny still decreases.

4 Strategy-2:

In this strategy we were supposed to use the `sections` construct of `OpenMP`.

4.1 Strategy 2.1

4.1.1 Implementation

In this strategy we created two sections that would execute in parallel. The first section consists of the L_3 which starts from `j+1` instead of `j` and the second section consists of the $(\texttt{i} = \texttt{j})^{th}$ iteration of L_3 peeled out, followed by L_5.

4.1.2 Handling Data Races

As seen in the previous strategy, that there was a dependency between L_3 and L_5 when `i=j` (As L_3 computes $L[j][j]$ and L_5 uses $L[j][j]$). We have handled that here by peeling out the first iteration from L_3 and adding it to the second section, just before L_5. We have kept the variables `sum, i, k` as private for the two sections. Now both these sections contain independent tasks, so for a given `j`, these tasks can be done in parallel.

```
#pragma omp section
{
    for (i = j + 1; i < n; i++) {
        ...
    }
}
#pragma omp section
{
    sum = 0;
    for (k = 0; k < j; k++) {
        sum = sum + L[j][k] * U[k][j];
    }
    L[j][j] = A[j][j] - sum;
    for (i = j; i < n; i++) {
        ...
    }
}
```

4.1.3 Analysis

Matrix Size	2 Threads	4 Threads	8 Threads	16 Threads
32	0.002417	0.002951	0.003008	0.004316
256	0.060544	0.056970	0.059781	0.062901
1024	1.411348	1.432234	1.552195	1.467119
2048	9.917850	10.037271	9.854597	10.088578
4096	128.893211	124.127724	128.653621	120.635376

Table 5: Time Taken (in seconds)

Matrix Size	2 Threads	4 Threads	8 Threads	16 Threads
32	0.594	0.487	0.477	0.333
256	1.123	1.193	1.137	1.081
1024	1.375	1.355	1.250	1.323
2048	1.573	1.555	1.584	1.547
4096	1.573	1.634	1.576	1.681

Table 6: Speed-Up

Matrix Size	2 Threads	4 Threads	8 Threads	16 Threads
32	0.297	0.122	0.060	0.021
256	0.561	0.298	0.142	0.068
1024	0.688	0.339	0.156	0.083
2048	0.786	0.389	0.198	0.097
4096	0.786	0.408	0.197	0.105

Table 7: Efficiency

Observations:

- For this strategy, there is no benefit in runtime and speedup on increasing the number of threads beyond 2 as the strategy uses only 2 threads. In fact the time taken slightly increases with the number of threads due to idle threads being present on the ready queue.
- The speedup and efficiency almost become constant on increasing the input matrix size

4.2 Strategy 2.2

Note that in the previous strategy, there was no extra improvement in case of more than 2 threads as there were only 2 sections which were executed serially. But now we can improve this by dividing the iterations between the threads manually, if there are more than 1 threads available in a section (It is a divide and conquer strategy).

4.2.1 Implementation

In this strategy, L_1 is executed serially, and each iteration of L_2 calls a recursive function which takes a few extra arguments.

```
void s2_crout_recurr(double const **A, double **L, double **U, int n, int num_threads, int j,
                     int start, int end)
```

`start` and `end` denotes the indices for the loops L_3 and L_5. This is a recursive function

- if the number of threads is 1: then execute the code serially
- if the number of threads if 2: then directly use strategy 2.1 for this case as there are only 2 sections in it.
- if the number of threads is greater than 3, then spawn 2 threads and divide the iterations between them, and recursively call this function again in 2 different sections. If p was the number of threads, then the first section gets $p/2$ threads and the second section gets $(p - p/2)$ threads.

4.2.2 Handling Data Races

Instead of adding the $(\texttt{i} = \texttt{j})^{\texttt{th}}$ iteration of L_3 before L_5 in the second section, we compute $L[j][j]$ at the start of each function call, so that this value is already computed before L_5 needs it.

4.2.3 Analysis

Matrix Size	2 Threads	4 Threads	8 Threads	16 Threads
32	0.004010	0.005124	0.007681	0.012351
256	0.063018	0.069797	0.079340	0.109429
1024	1.389561	1.053138	0.950852	1.086697
2048	8.547400	6.121328	4.687288	4.901045
4096	128.579722	71.527864	46.541525	41.844009

Table 8: Time Taken (in seconds)

Matrix Size	2 Threads	4 Threads	8 Threads	16 Threads
32	0.358	0.280	0.187	0.116
256	1.079	0.974	0.857	0.621
1024	1.397	1.843	2.041	1.786
2048	1.826	2.549	3.329	3.184
4096	1.577	2.835	4.357	4.846

Table 9: Speed-Up

Matrix Size	2 Threads	4 Threads	8 Threads	16 Threads
32	0.179	0.070	0.023	0.007
256	0.539	0.243	0.107	0.039
1024	0.698	0.461	0.255	0.112
2048	0.912	0.637	0.416	0.199
4096	0.788	0.709	0.544	0.303

Table 10: Efficiency

Observations:

- On smaller matrix size, there is no benefit in runtime as the synchronisation overhead is greater than the actual computation time. The speed-up < 1 in these cases. As we increase the number of threads, the runtime increases but the speed-up and efficiency decreases.
- On larger matrix size, the runtime decreases as we increase the number of threads. The speed-up > 1 and also increases but the efficiency still decreases.
- When computing on 16 threads the runtime is sometimes more than that when computing with 8 threads because of the limited number of threads available in out systems.

5 Strategy-3:

5.1 Implementation

Now as we had identified 2 independent section in strategy 2.1, the `for` loops in these sections, namely L_3' and L_5, execute in a serial manner. So there's still scope for more parallelization.
So as seen in strategy 1, L_1 can be parallelized straightaway using the `#pragma omp parallel for` construct. As for L_2, we initialize the `sections` construct with 2 threads, because there are only 2 sections.

```
for (j = 0; j < n; j++) {
    #pragma omp parallel sections num_threads(2)
    {
        #pragma omp section
        {
           ...
        }

        #pragma omp section
        {
            ...
        }
    }
}
```

The code inside these sections is same as strategy 3, but except now we wrap that code inside a function and pass the number of threads as a parameters in the function. Inside section 1, L_3 gets $p/2$ threads and inside section 2, L_5 gets $p - p/2$ threads, where p is the number of threads, given as an input parameter. So the workload of threads is approximately divided in half between these threads.

5.2 Handling Data Races

As we have just combined strategy 1 and 2 to get this strategy, the code does not have any data races.

5.3 Analysis

Matrix Size	2 Threads	4 Threads	8 Threads	16 Threads
32	0.002526	0.005126	0.006566	0.012668
256	0.059811	0.067457	0.075068	0.110074
1024	1.443843	1.109727	0.958791	1.059809
2048	10.080370	6.828592	5.369604	5.156743
4096	129.714238	74.199015	51.046069	44.918601

Table 11: Time Taken (in seconds)

Matrix Size	2 Threads	4 Threads	8 Threads	16 Threads
32	0.568	0.280	0.219	0.113
256	1.136	1.008	0.906	0.618
1024	1.344	1.749	2.024	1.831
2048	1.548	2.285	2.906	3.026
4096	1.563	2.733	3.972	4.514

Table 12: Speed-Up

Matrix Size	2 Threads	4 Threads	8 Threads	16 Threads
32	0.284	0.070	0.027	0.007
256	0.568	0.252	0.113	0.038
1024	0.672	0.437	0.253	0.114
2048	0.774	0.571	0.363	0.189
4096	0.781	0.683	0.496	0.282

Table 13: Efficiency

Observations:

- On smaller matrix sizes, the synchronization overhead is large, so the speedup < 1, and the time taken increases and speedup and efficiency decrease as the number of threads increase.
- On larger input sizes, the total time taken decreases as the number of threads increase. In these cases, the speedup is > 1 and it also increases with the number of threads, but the efficieny still decreases.
- As the problem size increases, the increase in speedup is very less, indicating that the overhead for this strategy is comparatively larger

6 Strategy-4:

In this strategy, we were supposed to use `MPI` for parallelising the program. Let n be the matrix size and p be the number of processes.

6.1 Implementation

The idea for this strategy is to divide the loop iterations between the processes, wherever possible. This can be done in 2 ways

- Divide the number of iterations in chunks of size n/p, and assign 1 to each process
- Divide the iterations in a round-robin fashion

The second one was much easier to implement as we only needed to if check $i\%p == r$, where i is the iteration number, and r is the rank of the process.
So, in this manner, the iterations of L_3 and L_5 were divided between the processes. Now as there is no shared memory between the processes, all of them have their own copies of L and U, so they would need to send the updated values to each other.
For this, after both L_3 and L_5, each processes broadcast the values it has computed to the other processes. The following happens for each iteration of L_2:

```
/* Loop 3 */
for (i = j; i < n; i++) {
        if(i % size == my_rank){
                ...
        }
}
for(i = j; i < n; i++){
        MPI_Bcast(&(L[i][j]), 1, MPI_DOUBLE, i % size, MPI_COMM_WORLD);
}

/* Loop 5 */
for (i = j; i < n; i++) {
        if(i % size == my_rank){
            ...
        }
}
for(i = j; i < n; i++){
        MPI_Bcast(&(U[j][i]), 1, MPI_DOUBLE, i % size, MPI_COMM_WORLD);
}
```

By doing an extra check in each iteration, we make sure that the iterations are divided between the processes in a round robin fashion.

6.2 Handling Data Races

As observed before, there is a data dependency between L_3 and L_5, so all the processes need to broadcast the updated matrix values after each of these loops.

6.3 Analysis

Matrix Size	2 Processes	4 Processes	8 Processes	16 Processes
32	0.058950	0.069593	0.077303	0.283989
256	0.126337	0.113075	0.131506	0.413885
1024	1.246939	1.311237	1.752982	3.558050
2048	13.279776	13.363491	14.775386	21.566459
4096	158.002372	131.515201	136.586873	152.400125

Table 14: Time Taken (in seconds)

Matrix Size	2 Processes	4 Processes	8 Processes	16 Processes
32	0.024	0.021	0.018	0.005
256	0.538	0.601	0.517	0.164
1024	1.557	1.480	1.107	0.541
2048	1.175	1.168	1.056	0.723
4096	1.283	1.542	1.484	1.330

Table 15: Speed-Up

Matrix Size	2 Processes	4 Processes	8 Processes	16 Processes
32	0.0126	0.0051	0.0023	0.0003
256	0.269	0.150	0.065	0.010
1024	0.778	0.370	0.138	0.034
2048	0.587	0.292	0.132	0.045
4096	0.642	0.385	0.186	0.083

Table 16: Efficiency

Observations:

- We know that in distributed programming the cost of communication >> cost of computation.
- On smaller matrix size, the runtime increases as we increase the number of processes and speedup << 1. As the runtime increases the speed-up and efficiency decreases.
- On larger matrix size, the runtime still increases as we increase the number of processes but the trend is not clear. The speed-up > 1 and decreases as the runtime increases. The efficiency also decreases as the runtime increases.
- For 16 processes, the runtime is always considerably higher because of the limited number of cores on our system. Oversubscribing leads to a sudden increase in the runtime.

7 Comparing the strategies

When looking at the results of all the 4 strategies, the following observations can be made:

- Strategy 4 is slowest of all the parallel strategies. This can be due to the fact that the processes spawned do not have shared memory, and inter-process communication is much costlier than intra-process communication. Also, creating other processes and allocating the matrices for each of them takes time
- strategy 3 is consistently slower than strategy $1, 2$, possibly due to more synchronization overhead by both `sections` and `pragma` constructs