COL380 A2

Rishabh Ranjan (2018CS10416), Daman Arora (2018CS50404)

April 2021

1 Introduction

In this assignment, we experiment with various strategies to parallelise Crout's decomposition of a matrix A into a lower triangular matrix L and a unit upper triangular matrix U such that $A = LU$. We use multi-threading with OpenMP and multi-processing with MPI.

2 Approach

2.1 Common

While most of the considerations are very different for the shared memory strategies using OpenMP and the distributed memory strategy using MPI, we have made some common optimizations which are described in this section.

2.1.1 Data Races

There are NO data races in all of our strategies. Distributed memory programs don't suffer from data races in general. For the shared memory programs, we have structured the parallelisation in such a manner that every memory location is written only by a single thread. Further, a memory location which is being written into is never read from by any other thread that may be executing concurrently. There is an explicit happens-before ordering before every write and the corresponding read. More details can be found in the section explaining the strategies.

2.1.2 Memory Layout

In all strategies we use 4 NxN (with N $\leq$ 5000) matrices A, L, UT and U. Here A is the input matrix. L and UT are the matrices used for computation and U is the transpose of UT which is finally output along with L. We keep all these as global 1D arrays with size fixed at 5000x5000.

This gives the following advantages:

1. Time is not wasted in malloc/calloc and free.
2. The arrays are allocated in the uninitialized data segment (BSS segment) which is guaranteed to be zeroed out by the OS. This is useful because L, UT and U are triangular matrices which have around half of the entries as zero.
3. The memory consumption is close to optimal because OS's such as Linux use zero-on-write for page allocation. This lets the OS not allocate physical memory for those parts which are not used by the program, like when using small matrices. Even with 5000x5000 matrices, many contiguous regions are zeros. Further in the case of the MPI strategy, these 4 matrices are declared on all the processes but the matrix U is used only by the one process which writes the output. In this case, we have observed significantly less memory consumption than the theoretical maximum required by 4 5000x5000 double arrays in 16 processes, pointing to the zero-on-write optimization being used profitably for the U matrices in the 15 processes which never reference U.

4. Using 1D arrays we can represent 2D matrices of any size $\leq$ 5000x5000 while still using contiguous memory locations which boosts cache performance.

2.1.3 Input:

We use fast IO with `fgetc_unlocked()` to avoid the expensive operation of acquiring the lock for every number scanned. This is safe because multiple read accesses do not constitute a data race. This gives us significant reduction in input times which reduces the serial part of the code and exposes a higher fraction of parallelisable code resulting in better speedups.

Please note that we have NOT tried to parallelise the I/O. This optimization affects the serial version in the exact same way as the parallel versions and is therefore gives speedups which are completely valid. Further, there were no restrictions against using fast input (which is common and also a good practice for programs like these where the input data is large). Moreover, the output times dominate the serial part anyway.

2.1.4 Conditional Compilation:

We use conditional compilation to obtain input, processing and output times (for this report) in debug mode (as this requires additional synchronisation with barriers for the observed times to make sense) but not in release mode which is used for competitive grading.

2.1.5 Dependency Analysis

On analysing the data dependencies with observe the following:

1. There exist loop carried dependencies (between entries of L and U) in the outer j loop.
2. This are no data dependencies between the inner i loop iterations for both L and U.
3. There exist a loop carried dependency in the sum used in the innermost k loop for both L and U.
4. The i loop for U depends on the value of L[j,j]. There is no other dependency between the i loop for both L and U.

We have handled the last data dependency (with L[j,j]) specially to get an i loop that is parallelisable. We don't try to parallelise the outer j loop or the innermost k loop, so these dependencies are preserved. Only parallelising the i loop is sufficient for getting satisfactory performance.

2.1.6 Guiding Principle

The guiding principle is to use minimal amount of OpenMP or MPI structures to avoid parallelisation overheads. One important optimisation in this regard is that we merge the two inner loops (one for L and the other for U) into one loop. This means that a single construct can be used to parallelise the merged loop, which is faster.

2.2 Shared Memory Strategies

2.2.1 False sharing and Cache performance

To avoid false sharing and to boost cache performance we use the transpose of U for computations. Note that the innermost k loop reads contiguous elements from L and UT (cache locality), while the writes to L and UT by the different iterations of the i loop (which are distributed over different threads) happen to locations far away (avoids false sharing).

2.2.2 Transformation

First we transform the program to remove the data dependency from L[j,j] to the i loop iterations of U. To do this, we shift the computation of L[j,j] such that whenever a value is calculated which affects L[j,j] it is accumulated into L[j,j]. Along with some pre-processing, this ensures that by the j-th iteration, L[j,j] is already at it's correct value and need not be calculated in that iteration.

The transformed program structure has 3 parts: a pre-processing step, a computation step and a post-processing step (transposing UT to get U). All these 3 parts are parallelisable.

2.2.3 Strategies

1. **Parallel For:** Here we use a parallel for loop with cyclic scheduling for pre- and post-processing (as they have linearly increasing workload with iteration number) and with the default scheduling to parallelise the merged i loop (as it has uniform workload across iterations).

2. **Parallel Sections:** We parallelise the for loops (the ones described above) by creating 16 sections (since we are guaranteed to have at most 16 threads) where each section is distributed the iteration numbers in a cyclic manner. We also tried putting the L loop and the U loop in different sections, but the current strategy gave better results.

3. **Parallel For + Parallel Sections:** We used parallel for to parallelise the i loop because it works better with the default scheduling of parallel for. We used parallel sections to parallelise the pre- and post-processing steps as cyclic scheduling is the ideal policy for them. We also tried nested parallelism and using for sections and for within the i loop but this did not give satisfactory results.

2.3 Distributed Memory Strategy

We observe the following for iteration j of the outer loop and iteration i of the merged i loop:

1. The computation of L[i,j] ($i \geq j$) requires L[i,k] and UT[j,k] ($k < j$).

2. The computation of UT[i,j] ($i \geq j$) required L[j,k] and UT[i,k] ($k < j$) and L[j,j].

So, we adopt the following strategy:

1. Every process computes L[i,j] and UT[i,j] for a fixed set of i's across the j iterations, particularly those i's for which i%p == r, where p is the number of processes and r in the rank of that process.

2. In the j-th iteration, the process in-charge of calculating for i == j broadcasts L[j,:] and UT[j,:] to all other processes.

3. The processes also need L[i,j] and UT[i,j] but because the same process calculates L[i,:] and UT[i,:], these values are already in its memory.

4. L[j,j] needs to be calculated beforehand. Observe that all the data required for it is already available to the process which in in-charge of i == j. So, this process calculates it while the L[i,j] and UT[i,j] broadcasts happen. When L[j,j] is computed, it is broadcasted separately.

5. A wait call is used to ensure that all the broadcasts have completed before proceeding to the calculations for the current iteration.

6. Finally when all iterations have completed, due to the series of broadcasts, all processes have the correct values for L[:,:] and UT[:,:]. Process zero then computes the transpose of UT to get U and outputs both L and U.

Note that using UT instead of U gives a boost to cache performance similar to the one observed in the shared memory case. But also, this lets us broadcast buffers which are contiguous regions in memory. This makes communication efficient.

3 Results

We conduct experiments for all 4 strategies for problem instances of sizes $1000, 2000, 3000, 4000$, and 5000. The number of threads we used in the experiments were 1, 2, 4, 8, and 16. All experiments were conducted on Intel(R) Xeon(R) W-2145 CPU @ 3.7 GHz. This CPU has **8 cores** and supports hyper-threading of 2 threads per core.

3.1 Strategy 1: Parallel For

	1	**2**	**4**	**8**	**16**
1000	1.08/1.00/1.00	0.86/1.26/0.63	0.73/1.49/0.37	0.67/1.62/0.20	0.65/1.67/0.10
2000	6.64/1.00/1.00	4.56/1.46/0.73	3.47/1.91/0.48	2.88/2.30/0.29	2.85/2.33/0.15
3000	20.04/1.00/1.00	12.88/1.56/0.78	9.15/2.19/0.55	7.29/2.75/0.34	7.16/2.80/0.17
4000	44.76/1.00/1.00	27.37/1.64/0.82	18.62/2.40/0.60	14.16/3.16/0.40	14.01/3.19/0.20
5000	82.74/1.00/1.00	49.35/1.68/0.84	32.70/2.53/0.63	23.99/3.45/0.43	23.69/3.49/0.22

Table 1: Total time(s)/Speedup/Efficiency

	1	**2**	**4**	**8**	**16**
1000	0.54/1.00/1.00	0.29/1.83/0.91	0.16/3.38/0.84	0.10/5.32/0.66	0.09/6.10/0.38
2000	4.40/1.00/1.00	2.28/1.93/0.97	1.17/3.75/0.94	0.59/7.39/0.92	0.58/7.62/0.48
3000	14.91/1.00/1.00	7.66/1.95/0.97	3.94/3.79/0.95	2.05/7.28/0.91	1.97/7.55/0.47
4000	35.51/1.00/1.00	18.00/1.97/0.99	9.26/3.83/0.96	4.81/7.39/0.92	4.63/7.67/0.48
5000	68.19/1.00/1.00	34.61/1.97/0.99	17.91/3.81/0.95	9.26/7.36/0.92	8.91/7.65/0.48

Table 2: Processing time(s)/Speedup/Efficiency

3.2 Strategy 2: Parallel Sections

	1	2	4	8	16
1000	1.09/1.00/1.00	0.84/1.30/0.65	0.71/1.54/0.38	0.65/1.69/0.21	0.66/1.67/0.10
2000	6.74/1.00/1.00	4.67/1.44/0.72	3.54/1.91/0.48	2.95/2.29/0.29	2.91/2.32/0.14
3000	20.18/1.00/1.00	12.93/1.56/0.78	9.20/2.19/0.55	7.36/2.74/0.34	7.24/2.79/0.17
4000	44.37/1.00/1.00	27.47/1.62/0.81	18.67/2.38/0.59	14.16/3.13/0.39	14.10/3.15/0.20
5000	82.29/1.00/1.00	49.40/1.67/0.83	32.63/2.52/0.63	24.02/3.43/0.43	23.86/3.45/0.22

Table 3: Total time(s)/Speedup/Efficiency

	1	2	4	8	16
1000	0.55/1.00/1.00	0.29/1.89/0.95	0.16/3.43/0.86	0.09/5.86/0.73	0.09/5.80/0.36
2000	4.49/1.00/1.00	2.34/1.91/0.96	1.20/3.73/0.93	0.66/6.82/0.85	0.63/7.17/0.45
3000	15.07/1.00/1.00	7.71/1.95/0.98	3.97/3.79/0.95	2.07/7.29/0.91	2.03/7.42/0.46
4000	35.19/1.00/1.00	18.12/1.94/0.97	9.31/3.78/0.95	4.82/7.30/0.91	4.74/7.42/0.46
5000	67.84/1.00/1.00	34.53/1.96/0.98	17.68/3.84/0.96	9.22/7.35/0.92	9.11/7.45/0.47

Table 4: Processing time(s)/Speedup/Efficiency

3.3 Strategy 3: Parallel For and Parallel Sections

	1	2	4	8	16
1000	1.09/1.00/1.00	0.84/1.29/0.65	0.71/1.53/0.38	0.65/1.69/0.21	0.66/1.66/0.1
2000	6.70/1.00/1.00	4.57/1.46/0.73	3.48/1.92/0.48	2.89/2.32/0.29	2.86/2.34/0.15
3000	20.24/1.00/1.00	12.92/1.57/0.78	9.24/2.19/0.55	7.26/2.79/0.35	7.16/2.83/0.18
4000	44.72/1.00/1.00	27.37/1.63/0.82	18.56/2.41/0.60	14.12/3.17/0.40	14.00/3.19/0.20
5000	82.37/1.00/1.00	49.35/1.67/0.83	32.56/2.53/0.63	23.88/3.45/0.43	23.71/3.47/0.22

Table 5: Total time(s)/Speedup/Efficiency

	1	2	4	8	16
1000	0.54/1.00/1.00	0.29/1.85/0.92	0.16/3.38/0.84	0.09/5.81/0.73	0.09/6.14/0.38
2000	4.46/1.00/1.00	2.29/1.95/0.97	1.19/3.75/0.94	0.59/7.49/0.94	0.58/7.64/0.48
3000	15.10/1.00/1.00	7.70/1.96/0.98	3.96/3.81/0.95	2.04/7.39/0.92	1.98/7.61/0.48
4000	35.55/1.00/1.00	18.02/1.97/0.99	9.22/3.86/0.96	4.79/7.42/0.93	4.63/7.68/0.48
5000	67.92/1.00/1.00	34.66/1.96/0.98	17.76/3.83/0.96	9.19/7.39/0.92	9.01/7.54/0.47

Table 6: Processing time(s)/Speedup/Efficiency

3.4 Strategy 4: Distributed Program in MPI

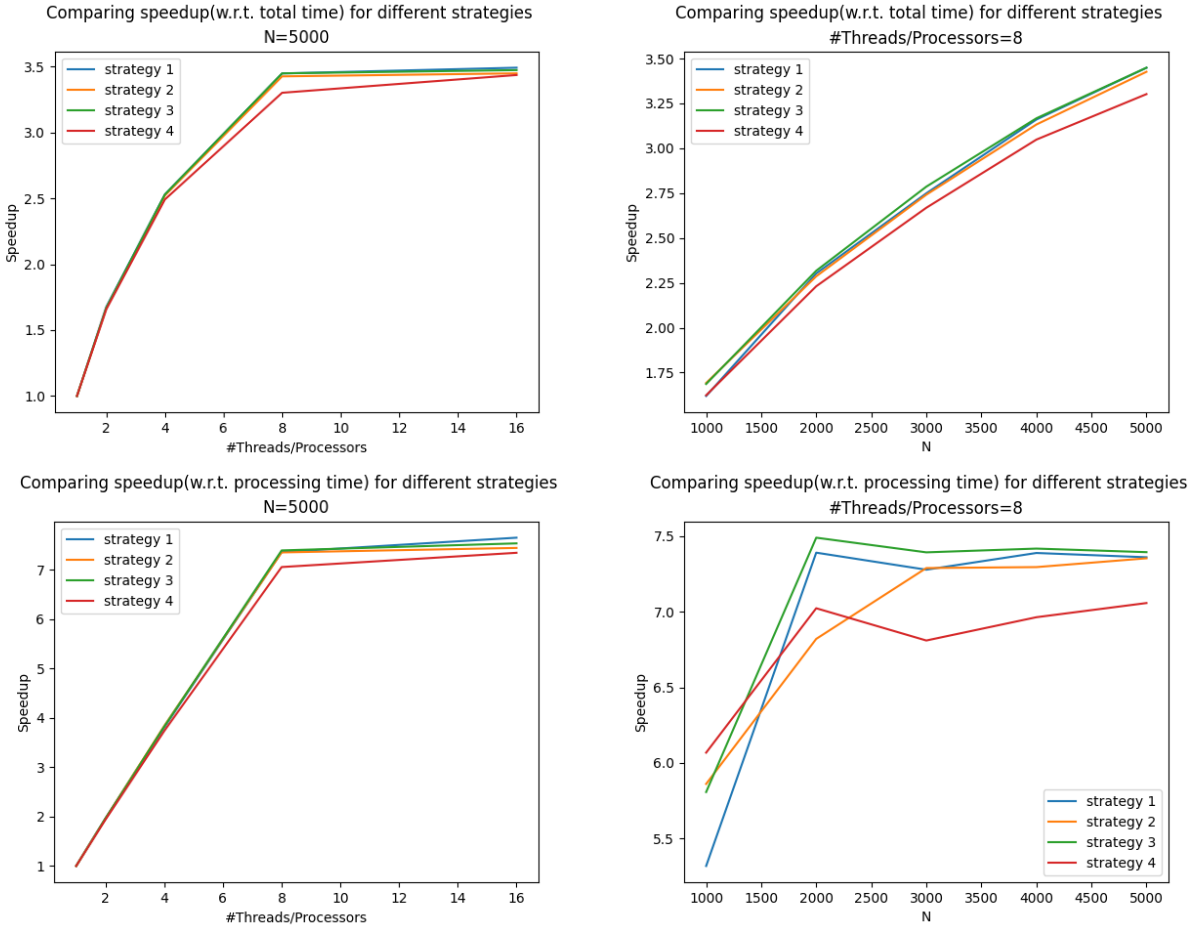
	1	2	4	8	16
1000	1.09/1.00/1.00	0.84/1.29/0.65	0.71/1.53/0.38	0.65/1.69/0.21	0.66/1.66/0.1
2000	6.71/1.00/1.00	4.65/1.44/0.72	3.53/1.90/0.48	3.01/2.23/0.28	5.05/1.33/0.08
3000	20.06/1.00/1.00	12.96/1.55/0.77	9.40/2.13/0.53	7.52/2.67/0.33	12.16/1.65/0.10
4000	44.34/1.00/1.00	27.43/1.62/0.81	19.01/2.33/0.58	14.55/3.05/0.38	23.32/1.90/0.12
5000	82.75/1.00/1.00	49.91/1.66/0.83	33.21/2.49/0.62	25.06/3.30/0.41	24.08/3.44/0.21

Table 7: Total time(s)/Speedup/Efficiency

	1	2	4	8	16
1000	0.54/1.00/1.00	0.28/1.90/0.95	0.15/3.48/0.87	0.09/6.07/0.76	0.10/5.68/0.36
2000	4.47/1.00/1.00	2.35/1.91/0.95	1.22/3.67/0.92	0.64/7.02/0.88	0.67/6.73/0.42
3000	14.98/1.00/1.00	7.76/1.93/0.97	4.11/3.64/0.91	2.20/6.81/0.85	2.16/6.92/0.43
4000	35.16/1.00/1.00	18.14/1.94/0.97	9.50/3.70/0.93	5.05/6.96/0.87	5.11/6.89/0.43
5000	68.27/1.00/1.00	35.20/1.94/0.97	18.27/3.74/0.93	9.67/7.06/0.88	9.29/7.35/0.46

Table 8: Processing time(s)/Speedup/Efficiency

4 Plots



5 Discussion

5.1 Trend with threads/processes

We observe that speedup w.r.t. the processing time is very close to the ideal value of the number of threads/processes for all strategies upto 8 threads/processes. For 16 processes the speedup increases but only slightly. This is easily explained by the fact that the CPU used for experiments has 8 cores with hyper-threading 2 threads per core. Of course, hyper-threading will not be able to match the speedup obtained from having real extra cores, especially in programs like these which are CPU-bound rather than IO-bound.

This shows that the implementations is highly efficient achieving close to optimal speedups for the parallelisable part. The gap from the ideal speedup is explained by the parallelisation overheads. The

speedups for the total time are reported for completeness. They show that the serial part of input and output significantly affect the speedups.

Among the different strategies, we find all of them give equivalent results, with the MPI strategy slightly slower than the OpenMP ones. This is to be expected because setting up processes and communicating between them is bound to be somewhat slower than using threads which all access the same memory.

Apart from speedup we have reported the times and efficiency of all experiments extensively. The high efficiencies also give confidence in our implementation. We also note that efficiencies decrease slightly with the number of threads/processes, which is again expected due to parallelisation overheads.

5.2 Trends with problem size

Speed up in the processing time saturates close to optimal at 2000 size matrices. However, the speedup in the total time keeps increasing with matrix size. This is because the I/O times increase quadratically with problem size whereas the processing times increase cubically, so the contribution of processing time (which is parallelisable) increases with problem size.

Again, the speedups are comparable (please take into account the scaling on the y-axis before judging the gaps in the plot), which the MPI strategy giving slightly worse results.

Efficiency increases with problem size (reaching near optimal for larger sizes) in all cases as expected.