COL380 : Parallel Programming

Parallel Crout Decomposition

Navneel Singhal 2018CS10360

Sarthak Agrawal 2018CS10383

April 2021

1 Algorithm Analysis

We highlight some salient features of the algorithm (the reference implementation) that we have utilized in reasoning about correctness of our parallel strategies.

Terminology We call the outermost loop as the **first nest**, the second level loops (there are 2 such loops) as **second nest** and finally the inner most loops (used to compute inner product) as **third nest**.

Loop Invariant of first nest After the i^{th} iteration of first nest, columns $1 \dots i$ of L have been computed correctly, and rows $1 \dots i$ of U have been computed correctly.

Loop dependency of first loop in second nest Computation of i^{th} column of L requires taking inner product of all rows of L with i^{th} column of U. Thus, this computation is dependent only on columns $1 \dots i-1$ of L and i^{th} column of U. Therefore there is **no** loop dependency in this loop and all iterations can be run in parallel. Note that i^{th} column of L may be computed without knowing i^{th} row of U except for the value $U[i][i]$ which is known to be 1.

Loop dependency of second loop in second nest Computation of i^{th} row of U requires taking inner product of all columns of U with i^{th} row of L. Thus, this computation is dependent on i^{th} row of L and rows $1 \dots i-1$ of U. Therefore there is **no** loop dependency in this loop and all iterations can be run in parallel Note that in particular i^{th} row of U may be computed without knowing i^{th} column of L except for the value $L[i][i]$.

Loop dependency of first nest From the above 2 observations in the second nest we may conclude that the i^th iteration of first loop depends on all the iterations $1 \dots i-1$ before it. Thus, there is a loop dependency here.

2 Approaches

2.1 Strategy 0 : Sequential Version

The implementation for this strategy is same as the reference implementation provided to us without any modifications.

The timing measurements for this strategy is given below. (Note that time is the wall clock time of the entire program, this includes, in addition to the core algorithm, the times for I/O (which is not insignificant).

For matrix with $N = 1000$ rows, time taken = 1.665 s
For matrix with $N = 5000$ rows, time taken = 255 s

2.2 Strategy 1 : Parallel For (OMP)

In this strategy as well as all the further strategies we first make the following optimization to improve caching performance of the program. We compute the transposed version of matrix U, and then at the end of compute take transpose once again to restore the original U. The advantage of this transpose is that when taking the inner product in the third nest, both L and U^T are accessed in row-major order, as opposed to column-major order in which U was being accessed in the reference implemented. Thus, this optimization makes the program more cache friendly.

Now, to parallelize the program, we first observed that it is not possible to parallelize the first nest (outermost-loop) as there is loop-carried dependency here that we laid out in the analysis above. But from the analysis we also observe that both the loops is the second nest may be parallelized as long as we run the second loop of second nest after the first loop of second nest. Therefore we place a parallel for construct before each of these loops.

Data Races In this version the variables A, L, U, j, n are shared among threads. Rest all are private variables. So any data race must be in the shared variables only. However observe that first of all A, j, n are only read-only variables so there can be no data-race in these. Secondly, in the first parallel for construct each thread would write to only $L[i][j]$ whose memory location will be different depending on j for each thread. Further as discussed in the analysis there is no loop dependency in this loop so there can be no data race in L. Now OMP automatically introduces a memroy barrier at the end of parallel for, so before entering U's loop all the threads must have computed their share of L so L must have been fully computed. Continuing by same logic we can see that there will be no data race in U also.

The timing measurements for this strategy is given below.

For matrix with $N = 1000$ rows

Number of Threads	1	2	4	8	16
Time Taken (s)	1.49	1.17	1.01	0.92	1.02
Real Speedup (incl. caching)	1.12	1.43	1.66	1.81	1.63
Actual Speedup	1.00	1.27	1.48	1.62	1.45
Efficiency	1.00	0.64	0.37	0.20	0.09

For matrix with $N = 5000$ rows

Number of Threads	1	2	4	8	16
Time Taken (s)	124	75	54	50	49
Real Speedup (incl. caching)	2.06	3.40	4.72	5.10	5.20
Actual Speedup	1.00	1.65	2.30	2.48	2.53
Efficiency	1.00	0.83	0.57	0.31	0.16

2.3 Strategy 2 : Parallel sections (OMP)

For this strategy the idea was to consider each loop of the second nest as one section and therefore run them in parallel. But there was a dependency between them. In particular computation of U required knowing value of $L[i][i]$ which is computed in the first loop of second nest. To satisfy this dependency we seperate the first iteration of the first loop (in the second nest) from the rest of the loop (as there is no dependency among iterations in the second nest). Thus, we have 2 sections now. In the first section we run all the iterations except the first one of the first loop in the second nest, and in the second section we run the first iteration that was remaining followed by all the iterations of U's loop.

Data Races In this version the variables A, L, U, j, n are shared among threads. Rest all are private variables. So any data race must be in the shared variables only. However observe that first of all A, j, n are only read-only variables so there can be no data-race in these. Secondly, by design there is no depenence among the 2 sections, as the first section writes to $L[i][j]$ with $i > j$ and the second section writes to $U[i][j]$ $i > j$ and $L[j][j]$ so there is no overlap. Further as shown in the analysis there is no dependence in either section.

The timing measurements for this strategy is given below.

For matrix with $N = 1000$ rows

Number of Threads	1	2	4	8	16
Time Taken (s)	1.49	1.27	1.30	1.26	1.13
Real Speedup (incl. caching)	1.12	1.32	1.28	1.33	1.47
Actual Speedup	1.00	1.17	1.14	1.18	1.31
Efficiency	1.00	0.59	0.29	0.15	0.08

For matrix with $N = 5000$ rows

Number of Threads	1	2	4	8	16
Time Taken (s)	126	80	81	83	83
Real Speedup (incl. caching)	2.02	3.19	3.15	3.07	3.07
Actual Speedup	1.00	1.58	1.56	1.52	1.52
Efficiency	1.00	0.79	0.39	0.19	0.09

2.4 Strategy 3 : Both parallel for and sections (OMP)

For this part we build upon the technique of both the previous parts. We first divide the second nest into sections as done in strategy 2, then inside each section we run the loop in parallel using parallel for construct. Here note that we have made use of nested parallelism in OMP. The division of threads is done as follows, in the outer parallel construct we spawn 2 threads (eliding the corner

case when num threads = 1), and then in each section we use nested parallel construct to spawn roughly $n/2$ threads per section.

Data Races In this version the variables A, L, U, j, n are shared among threads. Rest all are private variables. So any data race must be in the shared variables only. However observe that first of all A, j, n are only read-only variables so there can be no data-race in these. Secondly, by design there is no depenence among the 2 sections, as the first section writes to $L[i][j]$ with $i > j$ and the second section writes to $U[i][j]$ $i > j$ and $L[j][j]$ so there is no overlap. Further as shown in the analysis there is no dependence in the loop in either section so both the loops may be parallelized without any data race.

The timing measurements for this strategy is given below.

For matrix with $N = 1000$ rows

Number of Threads	1	2	4	8	16
Time Taken (s)	1.49	1.21	1.11	1.02	3.74
Real Speedup (incl. caching)	1.12	1.38	1.49	1.64	0.45
Actual Speedup	1.00	1.23	1.34	1.46	0.40
Efficiency	1.00	0.62	0.33	0.18	0.02

For matrix with $N = 5000$ rows

Number of Threads	1	2	4	8	16
Time Taken (s)	126	78	55	49	54
Real Speedup (incl. caching)	2.02	3.27	4.64	5.20	4.72
Actual Speedup	1.00	1.62	2.29	2.57	2.33
Efficiency	1.00	0.81	0.57	0.32	0.15

2.5 Strategy 4 : MPI

In our MPI implementation each process maintains its own local copy of all the three matrices - A, L and U. First, the master process (rank 0) reads matrix A from the input file and broadcasts it to all the other processes. Next, each process runs all the iterations of the first nest (as shown in analysis, first loop can not be parallelized because of dependency). However, each process only runs some iterations in the second nest depending on its rank. In other words, while in the case of sequential version in each iteration of first loop, one column of L and one row of U is computed, in the MPI version, each process only computes a contiguous chunk of that column of L and a contiguous chunk of that row of U. Once computed, every process shares its result to the rest of the processes and gathers results from the other processes. This can be done using the `gatherall` communication pattern. This way, although each process computes only a small part of L's column and U's row, at the end of this communication each process has result of full computation of L's column and U's row. At the end all the processes expect the master process exit which then prints the result to file and also exits.

Here, we experimented with some other communication patterns too. For example one possible alternative was that each process sends its result to the master, which would then assemble the final result and broadcast it to others for the next iteration.

Yet another strategy was to do all the communications fully asynchronously and in a non-blocking manner using MPI's immediate mode communication calls. However we found that the `gatherall` communication pattern was the fastest among the three so we use it in our final submission.

Data Race In this each process has its own memory, there is no shared memory so there is no possibility of data race except during communications. But we note that by default MPI gurantees that communication buffers will not be in data race (either by using blocking communication or by copying the buffer into system internal buffer) so therefore the entire program will be free of data race.

The timing measurements for this strategy is given below.

For matrix with $N = 1000$ rows

Number of Threads	1	2	4	8	16
Time Taken (s)	2.99	2.59	2.46	2.48	28.59
Real Speedup (incl. caching)	0.56	0.64	0.68	0.67	0.06
Actual Speedup	1.00	1.15	1.22	1.20	0.10
Efficiency	1.00	0.58	0.30	0.15	0.01

For matrix with $N = 5000$ rows

Number of Threads	1	2	4	8	16
Time Taken (s)	126.00	85.00	65.00	67.00	248.00
Real Speedup (incl. caching)	2.02	3.00	3.92	3.81	1.03
Actual Speedup	1.00	1.48	1.94	1.88	0.51
Efficiency	1.00	0.74	0.48	0.24	0.03