HW01 REPORT

March 26,2019

Yiğit Aras Tunalı
17519
Sabancı University
CS406 - Parallel Computing

Contents

0.1 Problem Statement

The problem to be implemented and parallelized is the permanent of a matrix. The algorithm we followed was the one proposed by Ryser but for the sake of better complexity, the Nijenhuis and Wilf version was implemented. Upon inspection of the pseudo-code delivered with the homework files, the main problem that can be seen is the dependency of X array to the previous one in each iteration. This X array needs to be updated during each loop and used for the calculation of the permanent value of p, which makes this task look not likely to be parallelized. The only location where the so-called dependency of X wasn't existent was the first iteration of the loop. So to be able to implement this algorithm in a parallel way, one has to break this dependency and introduce an independent way of updating X in each iteration of different threads.

0.2 Sequential Implementation

The first step I took was to implement the provided MATLAB code in a sequential manner. Apart from making some small C coding style improvements, I didn't spend too much time on the improvement of this implementation and the main goal was to just use this as a reference. The timings of this sequential code are as follows;

Table 1: Sequential Results

Opt Flag	Seconds
-O0	2.369
-O3	0.777

Every test that has been run for the results are done on the NEBULA machine. The used flags, environment variables, and compiler directives will be specified and the results are the average of 20 runs. Since the size of the Input4 is big enough to clearly show the changing trends in seconds between different results only the Input4 is used for the benchmarks.

0.3 First Parallel Implementation

Since the main problem to meet was to break the dependency on the X array, the first thing I tried to accomplish was to find a pattern in the Gray-Code sequence so I would be able to implement a X array calculation method for each different iteration of the main loop. Since for any i of the iteration, the formula is as follows $X_i = X_O + M[:, z_j]$ where j is the locations of all the bits that are active, i.e. 1, in Gray-Code of that specific i. The results of this parallelization are as follows (compiled with g++ -fopenmp and -O0 and -O3);

Table 2: First Parallel Imp. With Bad Complexity.

Flags	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
-O0	30.88	16.06	8.958	5.190	2.657
-O3	12.92	6.649	3.703	2.140	1.318

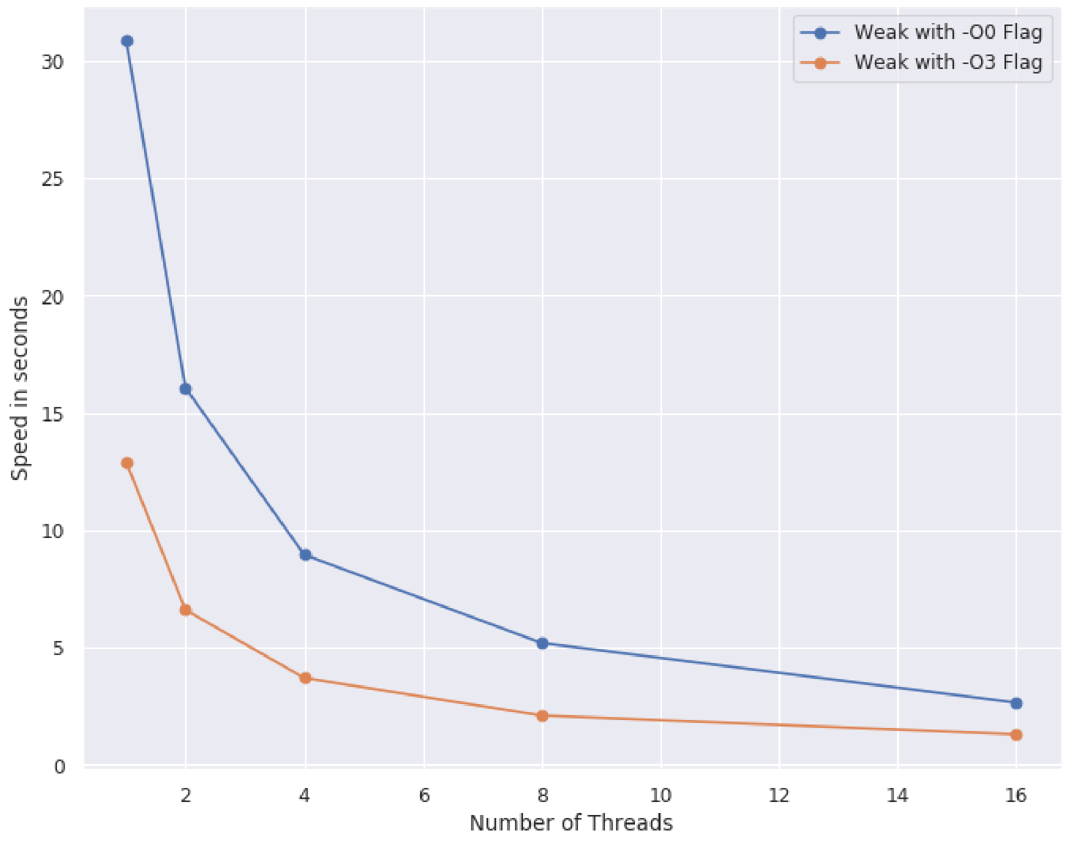

Figure 1: Comparison of -O0 and -O3 of the weak implementation

The reason why this approach didn't get good results is that while I tried to break the dependency on X array's values in each iteration, I increased the overall complexity to $O(2^n * n^2)$ just as in the naive implementation of the algorithm.

0.4 Better Parallel Implementation

The main idea in this approach is to partition the whole work into chunks. Since from the last method, I have realized I could calculate any X_i, I calculate the starting X_i vector for each chunk and continue to calculate the rest of the Gray-Code and the X array the same as sequential way. First the results for the implementation with no optimizations done;

Table 3: Better Parallel Implementation with no Optimization

Opt Flag	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
-O0	2.635	1.415	0.827	0.4587	0.2493
-O3	0.7502	0.446	0.234	0.133	0.097

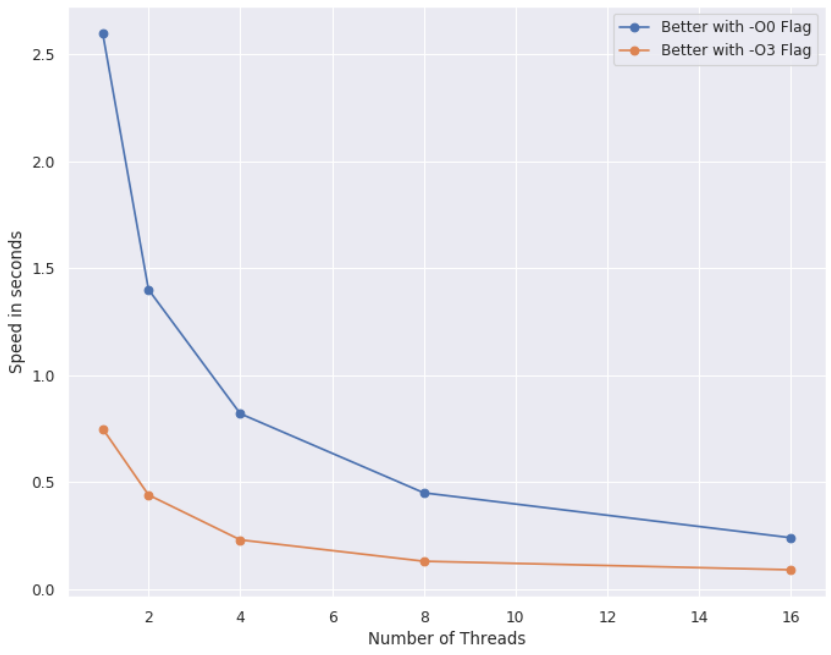

Figure 2: Comparison of -O0 and -O3 of the better implementation

Compared with the results from the weak parallel implementation from the last section, we can see there is a huge increase in times. At 16 threads this better implementation is more than $x10$ faster both in -O3 and -O0. Since the complexity is reduced as well the performance on lower threads are way better as well. By applying the higher complexity calculation a constant time at the beginning of each different thread and then continuing with the better complexity approach of calculating the changing bits between Gray-Code, we drastically improved the performance. After this point with lots of trial and error, I have also added more methods to further increase the performance.

0.4.1 Working on the Transpose Matrix

The first idea to come to mind was to work with the transpose of the matrix to increase spatial locality. This is done as a preprocessing and is just simply transposing the input into another matrix called MT, thus transforming column-order into row-order.

0.4.2 Getting Rid of log2 Function

Another subtle trick I have used was getting rid of log2 function to find the changing bit. Since it was a costly function, and the value I was checking only had 1 bit that would be 1, I used the built-in "__builtin_ctz()" function to get the location of that bit. After applying the transpose and getting rid of log2 the results were as follows;

Table 4: Better Parallel Implementation with transpose and ctz added

Opt Flag	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
-O0	2.606	1.364	0.814	0.441	0.229
-O3	0.672	0.377	0.209	0.127	0.091

As can be seen from the results the improvements are not major and just slight advances on the first approach and it's results. Since these approaches were more on the programming side (apart from the spatial locality gained by transpose), the next idea for an approach was to use different OpenMP methods that would utilize the hardware as well. The graph of Table 4 results are also below;

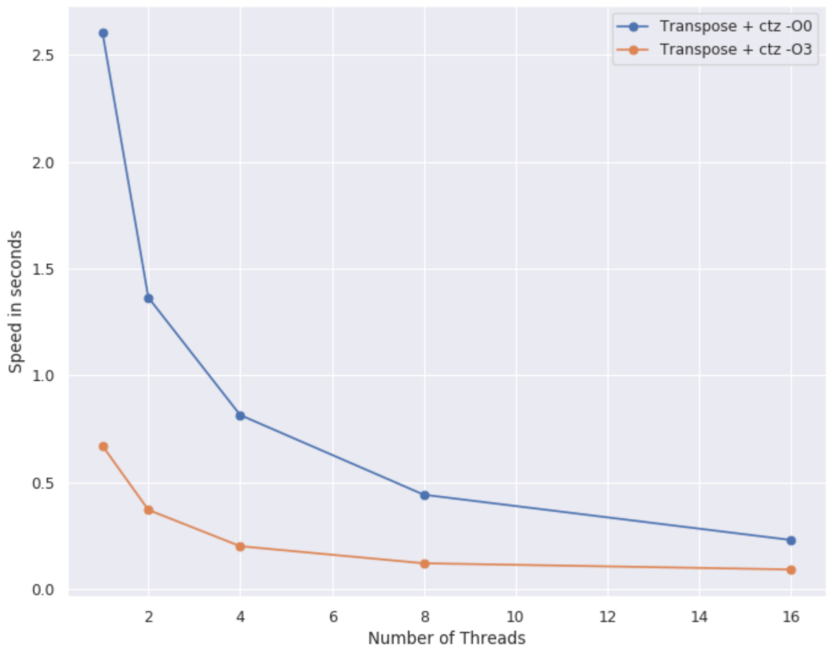

Figure 3: Comparison of -O0 and -O3 of the better implementation with transpose and ctz

0.4.3 Usage of NUMA

The next idea of improvement was to use the NUMA structure. With using the "proc_bind()" pragma of OpenMP, I first tried to spread and closing arguments, spread worked better. Then the option was either OMP_PLACES=cores or OMP_PLACES=sockets. Sockets option performed badly, probably because of the communication overhead between the sockets. The cores option worked really well and it gave me the most improvement overall from the approaches I tried.

0.4.4 Usage of SIMD

Final improvement technique I used was to use "SIMD" pragma of OpenMP on an inner loop of my parallel region, where the value that will be added or subtracted from p will be calculated for the specific i'the iteration.

0.4.5 Final Results of all the Improvements

Table 5: Better Parallel Implementation with all improvements added

Opt Flag	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
-O0	2.334	1.143	0.721	0.377	0.193
-O3	0.536	0.256	0.142	0.082	0.046

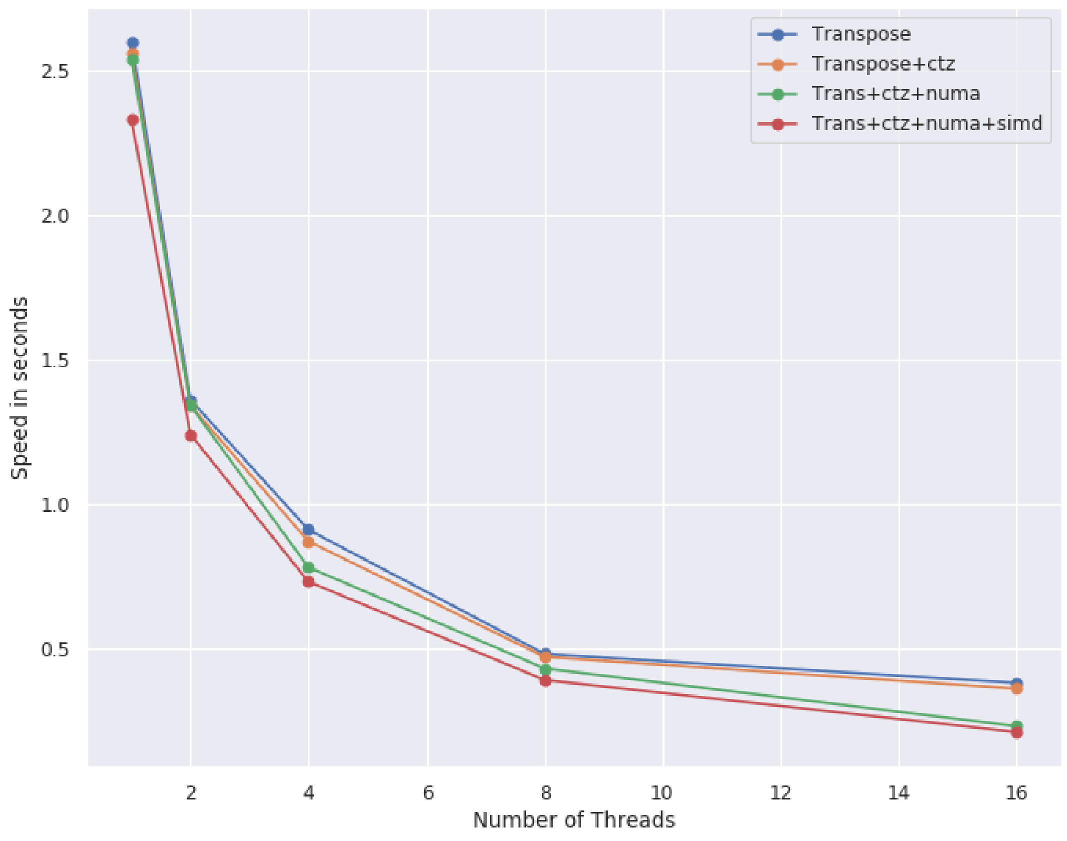

Figure 4: Comparison of all methods with -O0

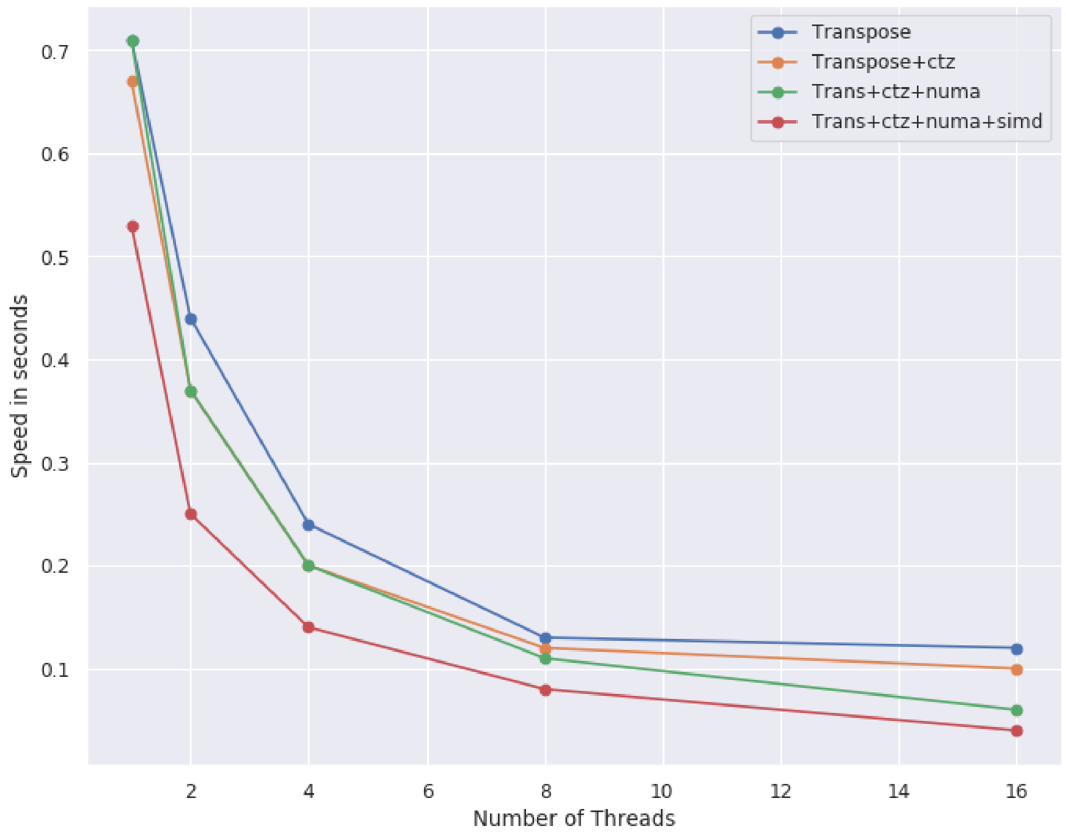

Figure 5: Comparison of all methods with -O3

0.5 How to Run the Final Code

The final code is the better version of the parallel algorithm with all the improvement techniques listed above as subsections. Before running the code one has to set the environmental variables; "export OMP_PLACES=cores" has to be run on the terminal. The compiler flags are as follows as "g++ -fopenmp -O3 src.cpp -mavx -mavx2".