Assignment 1

SPM course a.a 23/24

Spiridon Dragoș, Erasmus Student

d.spiridon@studenti.unipi.it

Problem:

Suppose you want to parallelize an algorithm that exhibits an upper-triangular wavefront computation pattern on an NxN matrix. Each element of an upper diagonal (starting from the leading diagonal) can be computed independently. Instead, distinct upper diagonals have to be computed serially in order (first, all elements of the diagonal k, and once the computation has completed, all elements of the diagonal k+1).

Solution and possible solutions:

For a first solution, we will take the original code ("wavefront") and try to split the main diagonal into the work for multithreading. For example, we will consider the first diagonal to be split with block mechanism, which will separate it by chunks in the most equal way. The split is made by this code:

```
uint64_t modulo = (N - k) % actual_num_threads;
uint64_t add_start = std::min(modulo, id);
uint64_t add_end = std::min(modulo, id + 1);
uint64_t start = (N - k) / actual_num_threads * id + add_start;
uint64_t end = (N - k) / actual_num_threads * (id + 1) + add_end;
```

In which "start" and "end" variables represent the starting and ending point from where we will take the values from the main diagonal. Another solution is to apply the cycle mechanism, which will simplify the splitting between the threads, due to only modifying the "for" which will go through the diagonal.

A solution which I do not think it would be implementable but it may possibly work is based on the work duration of each independent task. The idea is to split the diagonal task approximately equal in terms of their sum instead of using a specific mechanism, as we know that the work time of each matrix value is calculated by its value. The idea is not implementable unfortunately as it can take a lot of time to sort them through this process and also there is no (from my findings) algorithm which does that, only brute force.

For the last diagonals, where the number of threads is smaller than the diagonal length, for the sake of simplicity (and for not creating more barriers for each case), the threads which would not get any data to "work" will only go directly to the barrier and wait until the others will finish their job. For example, in Figure 1, the matrix has N=10 size and 4 threads are running through the matrix. The thread 4, when it arrives to 8^{th} diagonal (N – thread_num + 2 or 10 – 4 + 2), the number of threads is higher than the dimension of the diagonal (4 > 3), and so the thread goes directly to the barrier and waits for the other 3 threads to finish their job. The same thing also happens to the cycle separation of data through the diagonal.

Compile and run:

This code was run on the university cluster, which is spmcluster.unipi.it. For compiling the code, it is necessary to use the "include" folder given by the teacher with the hpc_helpers.hpp file in it. The command for running the code is:

```
g++ -std=c++20 -O3 -march=native -Iinclude  UTWavefront.cpp -o UTW
```

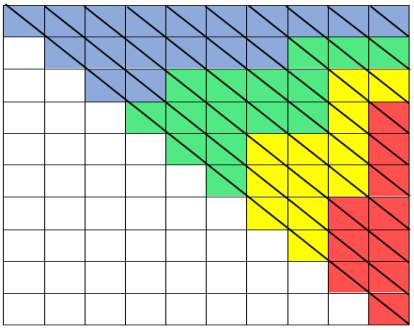


Figure 1. The works of multithreading by the block

This command will create an executable file “UTW”. For compiling it, you can use:

```
./UTW 512 1 1000 8 0
```

Where:

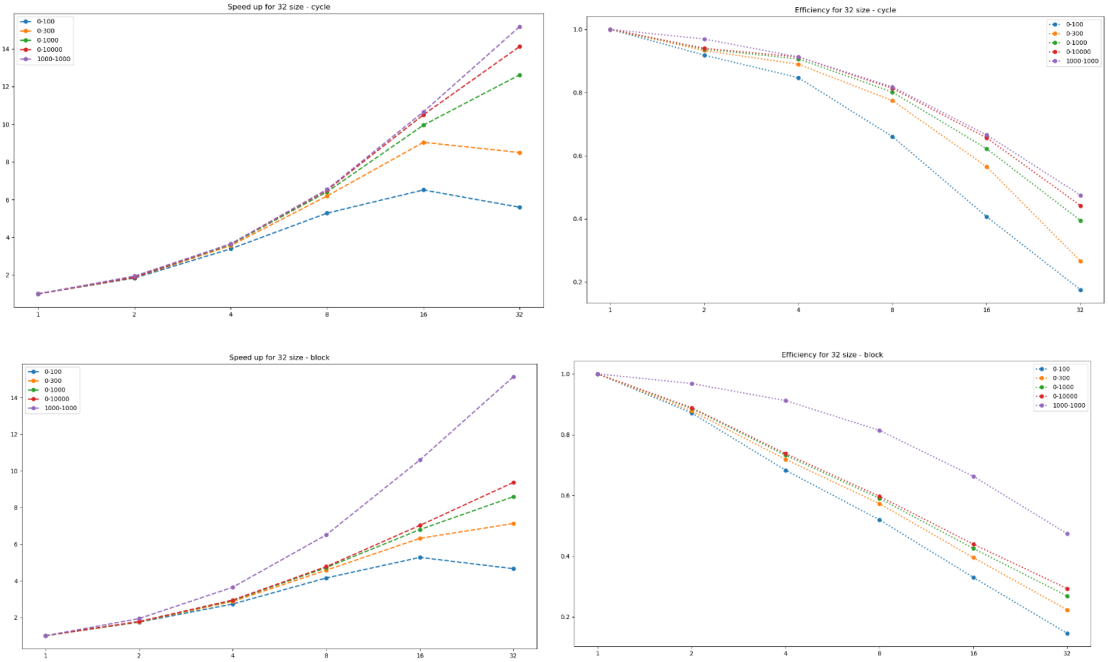
- 1st element: matrix size
- 2nd and 3rd element: minimum and maximum randomly generated (assigning the work time)
- 4th element: number of threads
- 5th element: cycle (0) or block (1)

For running with more threads and saving the results in a file (test.out for example) you can use this command:

```
for i in 1 2 4 8 16 32 64 128 256 512; do ./UTW 512 100 100 $i 0; done | sed -n -E
"s/^([0-9]+).* ([0-9]+\.[0-9]+).*$/\1 \2/p" > test.out
```

Results – strong vs weak scaling:

For testing the task, we will run the code with two matrix sizes: N = 32 and N=1024, in terms of **strong scaling**, with different values for min and max. We will plot only the speed-up, with the formula $T(N,1)\ /\ T(N,p)$, and the efficiency with $speedup/p$, and also testing for both cycle and block. Let’s start for matrix size equal to 32:

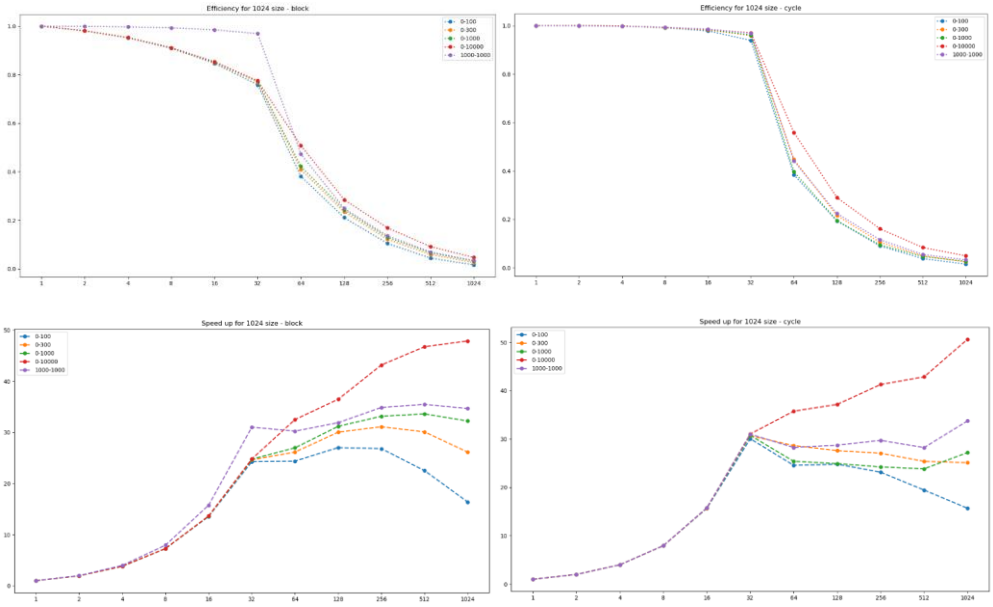


Threads	1 (seq)		2		4		8		16		32	
Sliced with	cycle	block	cycle	block	cycle	block	cycle	block	cycle	block	cycle	block
0-100	0.0272	0.0270	0.014	0.015	0.0080	0.0099	0.0051	0.0065	0.0041	0.0051	0.0048	0.0058
0-300	0.0806	0.0805	0.043	0.045	0.0226	0.0280	0.0130	0.0175	0.0089	0.0127	0.0094	0.0112
0-1000	0.2675	0.2674	0.142	0.150	0.0738	0.0913	0.0417	0.0566	0.0268	0.0393	0.0211	0.0311
1000-1000	0.5286	0.5284	0.272	0.272	0.1447	0.1448	0.0808	0.0811	0.0496	0.0498	0.0348	0.0348
0-10000	2.6700	2.6700	1.420	1.502	0.7315	0.9050	0.4105	0.5590	0.2542	0.3800	0.1889	0.2847

For a small matrix size, we see that the speedup is increasing approximately linearly by the size of the task. If the work has a low duration of time, then at the last thread it will go worse than before, but if the work has higher and higher duration, then the process will become more and more linear. From the perspective of efficiency, it still decreases linearly, and goes near 0% when it arrives to the number of threads equal to the matrix size. It makes sense because in each step there are a lot of threads and most of them only goes directly to the barrier.

From the perspective of strong scaling (horizontal lines in the table), the parallelized program creates a minus log form which tells us that it actually reduces the time by using more threads (not in all cases though, like using only work between 0-100, where it actually increases the time at the end).

And now, for matrix size equal to 1024:



Threads	1 (seq)		8		32		128		512		1024	
Sliced with	cycle	block	cycle	block	cycle	block	cycle	block	cycle	block	cycle	block
0-100	26.250	26.252	3.310	3.6124	0.8736	1.0819	1.0592	0.9719	1.3503	1.1660	1.6752	1.6018
0-300	78.693	78.693	9.918	10.805	2.5654	3.1969	2.8523	2.6200	3.1011	2.6152	3.1349	3.0125
0-1000	262.23	262.23	33.04	35.979	8.5302	10.598	10.526	8.4128	11.005	7.8053	9.6401	8.1402
1000-1000	524.83	524.83	66.05	66.058	16.943	16.923	18.277	16.459	18.604	14.7989	15.549	15.145
0-10000	2622.0	2622.0	359.6	359.67	105.75	105.75	71.921	71.921	56.109	56.109	54.774	54.774

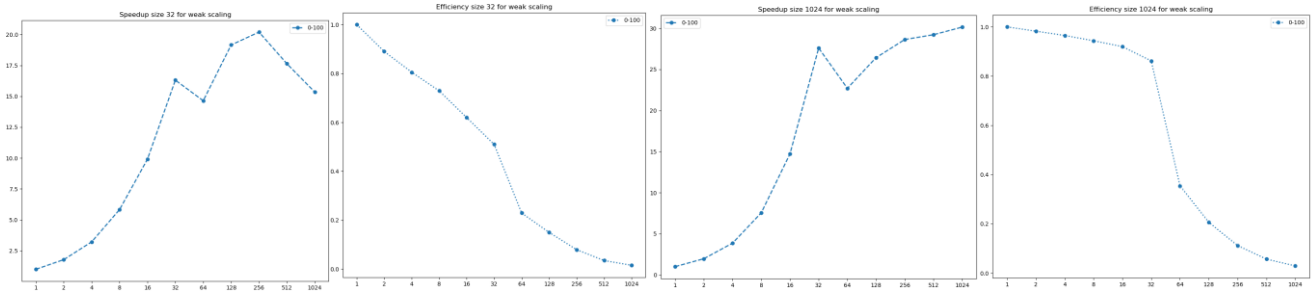
In this case, the efficiency of the program is going drastically low after more than 32 threads are used. Even though the speed up seems to increase, it is not scalable after the usage of more than 32 threads. In

both graphs, it drops from near 90% efficiency to 40-60%, and even lower afterwards until it arrives close to 0% efficiency. In conclusion for the strong scaling, only until 32 threads the scaling is actual good, after that, it starts acting abnormally (probably due to cores and too many threads/ the use of barriers).

For the **weak scaling** case, it will be analyzed only for N=32 matrix size (for the sequential) using only the block mechanism. For the experiment to be correct, when we multiply the number of threads, the matrix will be multiplied by $\sqrt{2}$, because the complexity of solving the matrix it quadruples if it is multiplied by two. For example, for thread 4 we will have 64x64. For thread 8, we should not have 128x128, as it is multiplied by 4from the perspective of 64x64, not by two, so the values chosen will be 90x90 (90x90=8100, 64x64x2=8192, close enough). The matrix size table for each thread will be (also with their times):

Num threads	1 (seq)	2	4	8	16	32	64	128	256	512	1024
Size: 32*$\sqrt{p}$	32	45	64	90	128	181	256	362	512	724	1024
T(32*$\sqrt{p}$,1)	0.0275	0.052	0.010	0.207	0.417	0.832	1.665	3.316	6.619	13.23	26.48
T(32*$\sqrt{p}$,p)	0.0275	0.029	0.032	0.035	0.042	0.051	0.113	0.173	0.327	0.750	1.726
Size: 1024*$\sqrt{p}$	1024	1448	2048	2896	4096	5792	8192	11585	16384	23170	32768
T(1024*$\sqrt{p}$,1)	26.516	52.941	105.93	211.79	424.07	847.06	1694.4	3388.8	6778.2	13555	27112
T(1024*$\sqrt{p}$,p)	26.516	26.939	27.441	28.053	28.812	30.704	74.659	128.26	236.79	464.02	899.66

And results (speed-up $T(32 * \sqrt{p}, 1) \ / \ T(32 * \sqrt{p},\ p)$, and efficiency $speedup \ / \ p$):



Until observing the results, we observe that the computation with multiplying with the square root of 2 is working properly as we see that the time is approximately doubling. From the results perspective, again, we meet the same problem as before: by accessing more than 32 threads, the efficiency drops significantly and the speedup is worse after using 64 threads and more. As before, the efficiency drops to 0%, and also the speedup seems that it is increasing until 256 threads, were

Conclusion

The separation of the working threads is splitting the data approximately equally, even though in terms of globally splitting the data equally, the separation is not well made. The focus on the splitting the data was on the diagonal data, and not on the entire number of data.

In terms of strong and weak scaling, we observe that both of them have a good efficiency until the number threads exceed 32. In the case of a lower matrix and not exceeding 32 threads, the efficiency decreseases linearly, from 1 to 0.5, whereas in the case of a higher matrix, the efficiency still decreases linearly, but it goes from 1 to approximately 0.9 (except for block efficiency, without 1000-1000 case), which denotes that by increasing the matrix size, the efficiency will be better by enlarging the number of threads, no matter the scale we use.