Description of implementations

For both V1 and V2 solutions, I implemented cache-aware algorithms to exploit the temporal and spatial locality of the data to make better use of the cache memory and increase the performance of my solutions.

In the problem of finding the longest common subsequence, the value of each element is dependent on the value of the element to its left and to the element above. Scanning anti-diagonally helps us make use of tasks and compute each element on the anti-diagonal in parallel.

V1 with explicit tasks

The algorithm is the following:

- Iterate the anti-diagonals of the matrix.
- Find the bottom and the top blocks in the antidiagonal.
- Create tasks for each block with dependencies.
- Traverse the blocks.
- Process the elements inside a block.

Dependencies

As one task has to wait until other tasks that are processing the blocks to the left and above are done, synchronization between them is needed. I set the currently running task dependent on the last element of the block to the left and the last element of the block above by defining an input dependency. I also defined an output dependency to make other tasks dependent on the current processing block's last row and last column.

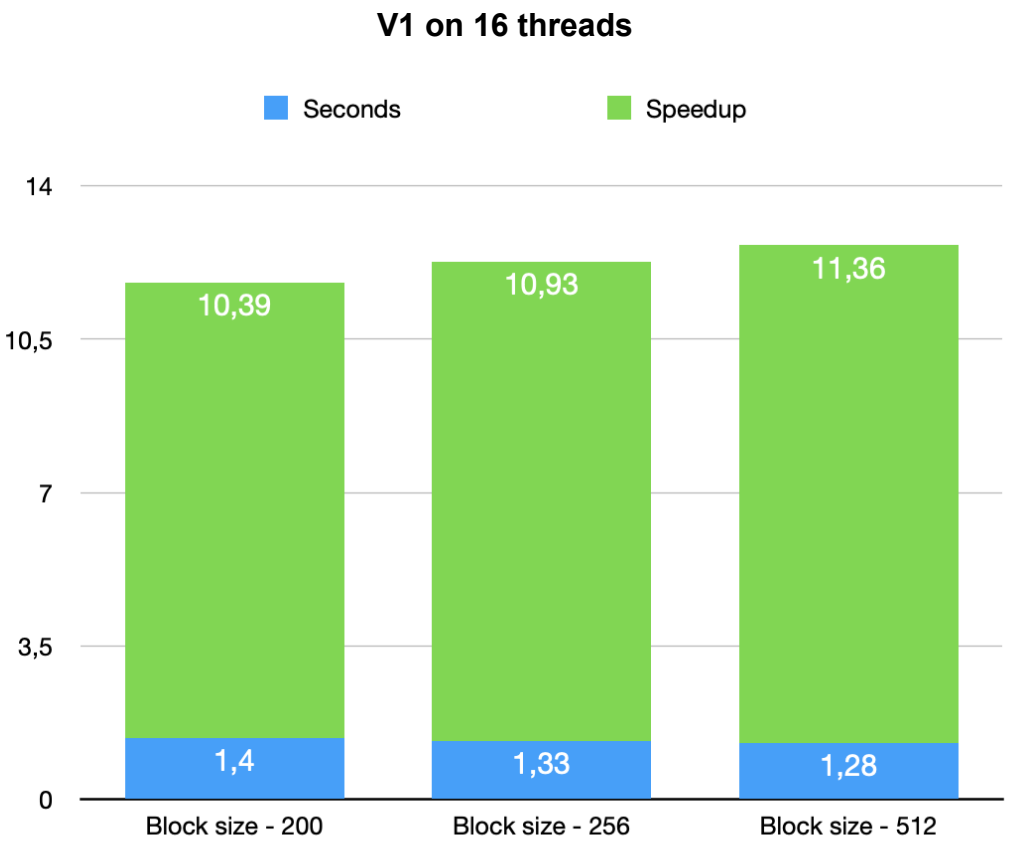
Data Scoping

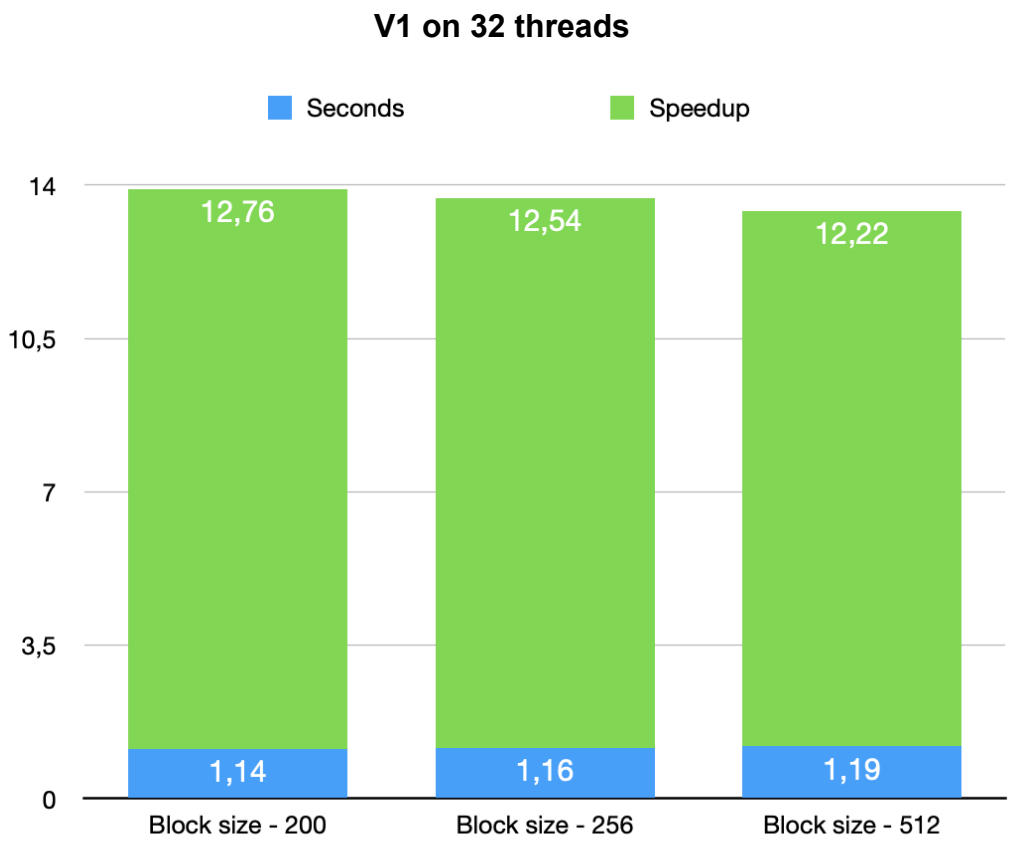
On the task construct, I defined the following variables as firstprivate: r*ow, col, count*.
I defined these variables as *firstprivate* in order to make sure that these values will not be affected by other tasks and that the current task will have its own predefined local copy of them.
r*ow, col* variables are for traversing the blocks. As *entries_visited* variable has to be updated synchronously by each task, I made the count variable local to the currently running task so that when the traversal of a block is over it will update the *entries_visited* variable in a synchronous way, without causing poor performance.

Granularities and the number of tasks

The generated amount of tasks depends on the number of blocks in an antidiagonal, which in its turn depends on the size of a block. The amount of elements that a task is performing is dependent on the defined size of the block.
To change the amount of work that a task is performing I tried different values of block sizes.

All speedups are in relation to the serial version (14,55 seconds).





V2 with taskloop

The algorithm is the following:

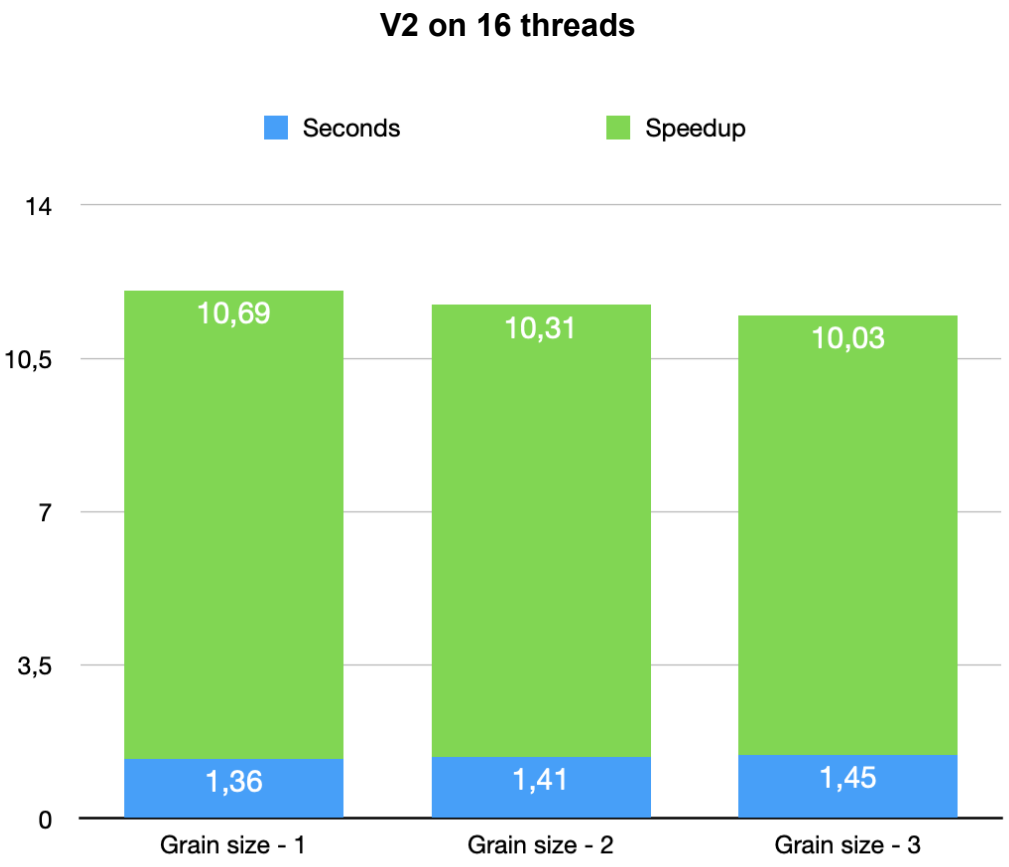
- Iterate the anti-diagonals of the matrix.
- Find the bottom and the top blocks in the antidiagonal.
- Iterate the blocks in the antidiagonals using taskloop.
- Traverse the blocks.
- Process the elements inside a block.

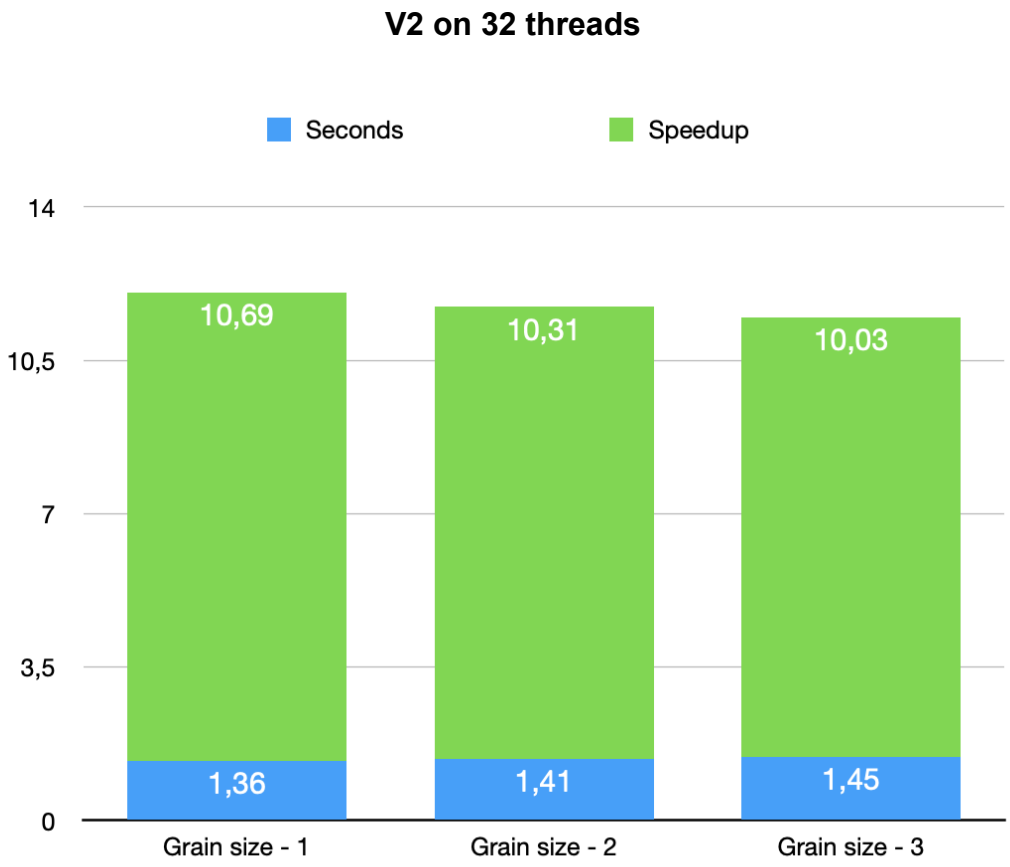
Data Scoping

For taskloop I defined the following variables as private: r*ow, col.*
In order to make sure that each task created inside the loop has its own copy I defined them as private.
To synchronize update for *entries_visited* variable I used *reduction* clause.

Granularties and the number of tasks

The amount of tasks created depends on the size of the block and the specified grainsize.
After trying different sizes of blocks, I chose 256 as the block size, and to specify the number of blocks each task processes I used the *grainsize* clause.
For example for grainsize of 1 and block size of 256, each task will process one block and the number of tasks will be 200.





Observations

The V1 implementation with explicit tasks overall produces better performance than V2 created with taskloop. The reason for that is that explicit tasks are more flexible in the means of synchronization by allowing to define dependencies between tasks. Taskloop on the other hand works as a *taskgroup*. In V2 it processes all blocks in an antidiagonal as in taskgroup, waits until they are all processed, and then proceeds to the next antidiagonal's blocks. In V1 tasks don't wait for all the elements in an antidiagonal to be processed and then start the next. Instead, dependencies between tasks are defined which makes better use of parallelism.