Arquiteturas de Alto Desempenho-Project 2

Sorting Sequences of Values

Prof. António Rui Borges

TP2 - Grupo 2
Rúben Castelhano, 97688
Carlos Costa, 88755

incOrderRow - analysis

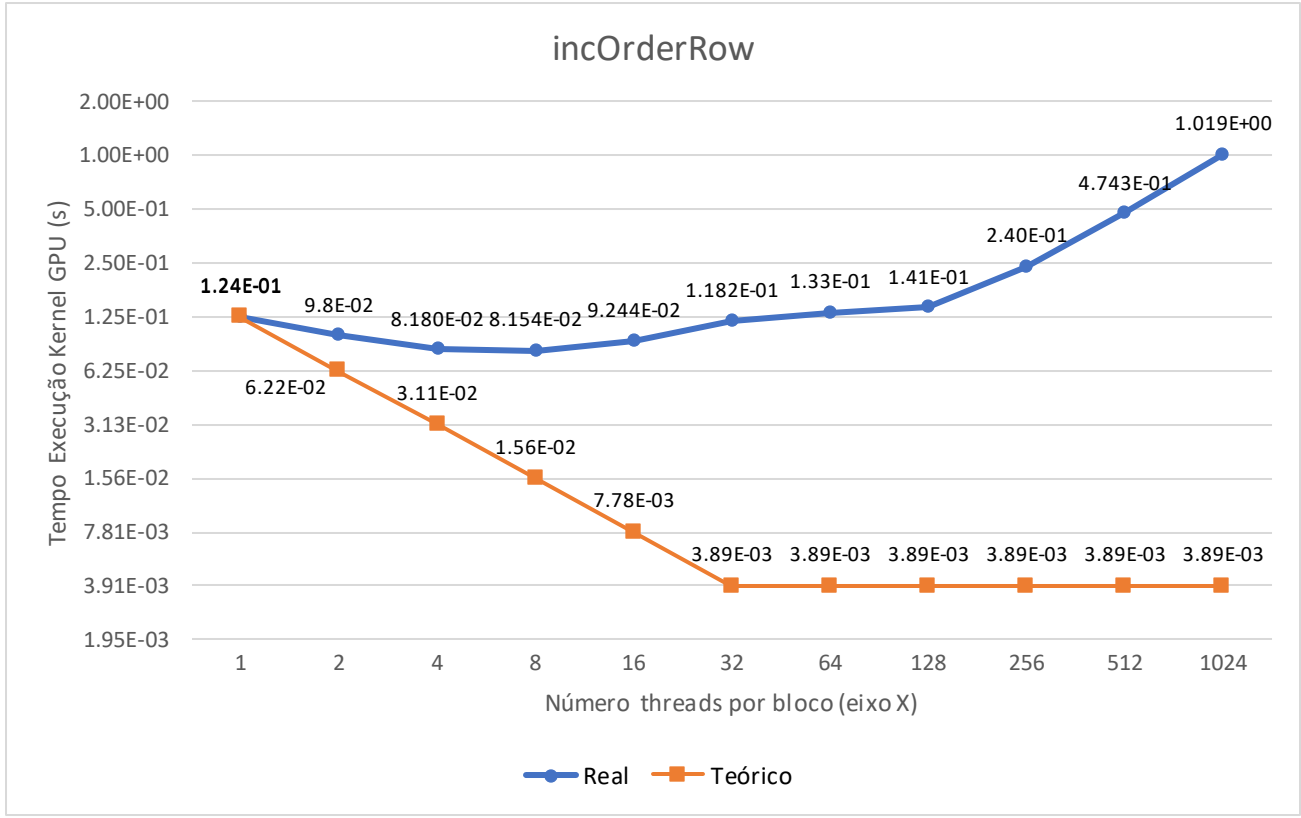
As a first approach we varied the number of threads per block on the x axis and created the following graph for analysis. We concluded that initially, for valued up to 8 threads per block, the execution time decreases, although much less than we predicted. From 16 up to 128 threads per block the execution time suffers a significant increase and from 256 up to 1024 threads per block the execution time gets approximately 2x worse.

The way the data is organized in this program, leads to inefficient access by multiple threads, due to the fact sequential threads will access noncontiguous memory sections which will result in more memory transactions and more cache thrashing.

From 2 up to 8 threads per block the benefit of running more threads in parallel outweighs the memory and cache miss penalties.

From 16 to 1024 threads the memory transaction and cache miss penalties outweighs any benefit gained from running more threads in parallel. Is should be noted that from 64 threads per block (results in 16 blocks) and onwards, full GPU occupancy is no longer possible because the GPU used has 24 SMs and because all threads in a block are run on the same SM.

Best Value: 8 Threads per Block.



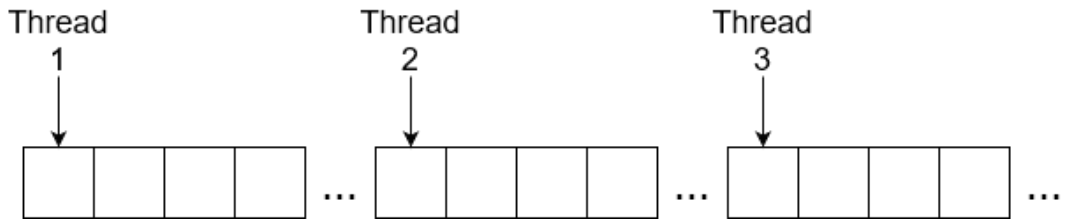


Figure 2: Memory access scheme

incOrderRow - analysis

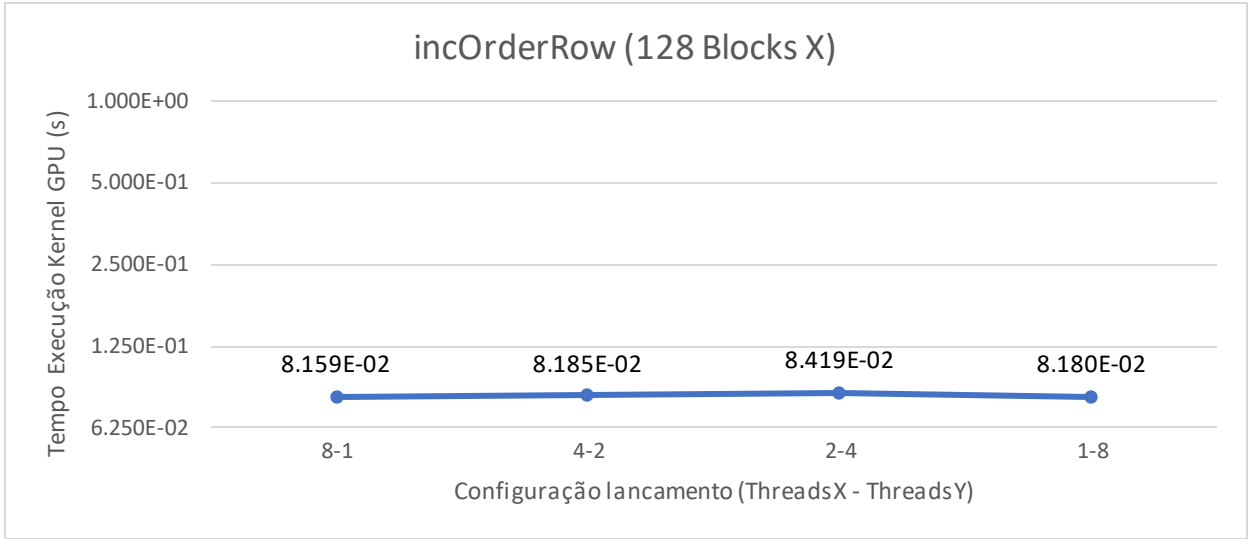
After determining that 8 threads per block offered the best performance, we varied the Block launch configuration on the X and Y axis. The original configuration of 8 Threads on the X axis per block continued to offer the best performance followed by the 1-8, 4-2 and 2-4 configuration, respectively.

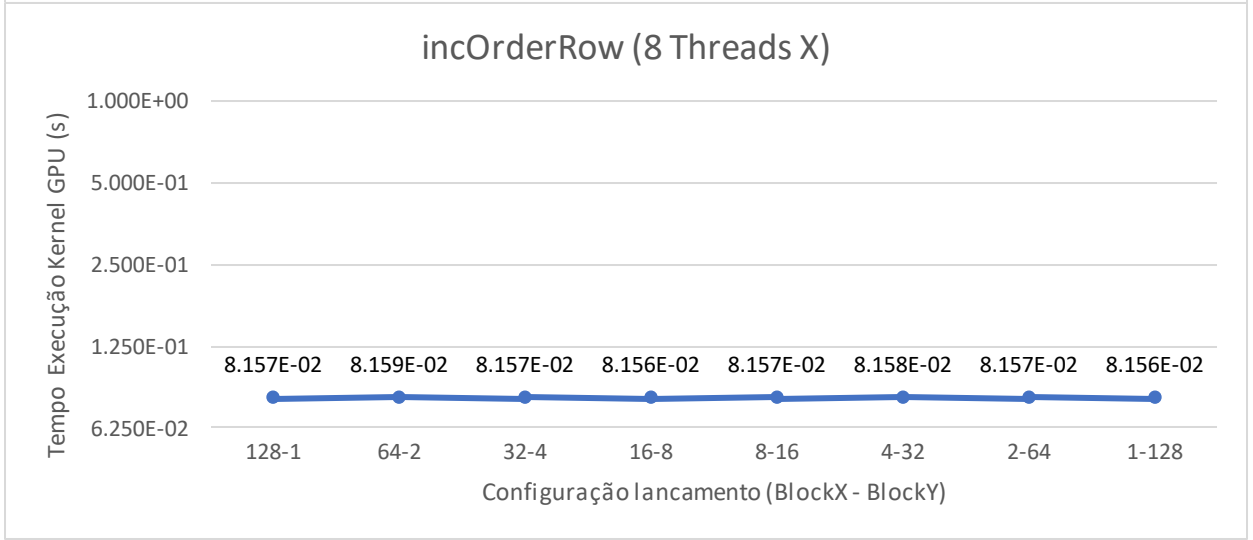
We then varied the Grid Launch Configuration on the X and Y axis maintaining the best Block configuration. The performance variation observed was minimal, although best performance was obtained with the 16-8 and 1-128 Grid configuration. Since every thread in a block is scheduled to the same SM, a variation of grid dimensions resulting in the same Block count should offer little to no performance advantage which coincides with what we observed.

Best Configuration: Grid of 16x8 Blocks of 8 Threads.

Average CPU time: 6.1E-01 seconds

Conclusion: Although the way the memory is mapped is far from ideal, and the program execution time doesn't scale with increased parallelization although it still performed better than the CPU sequential version.



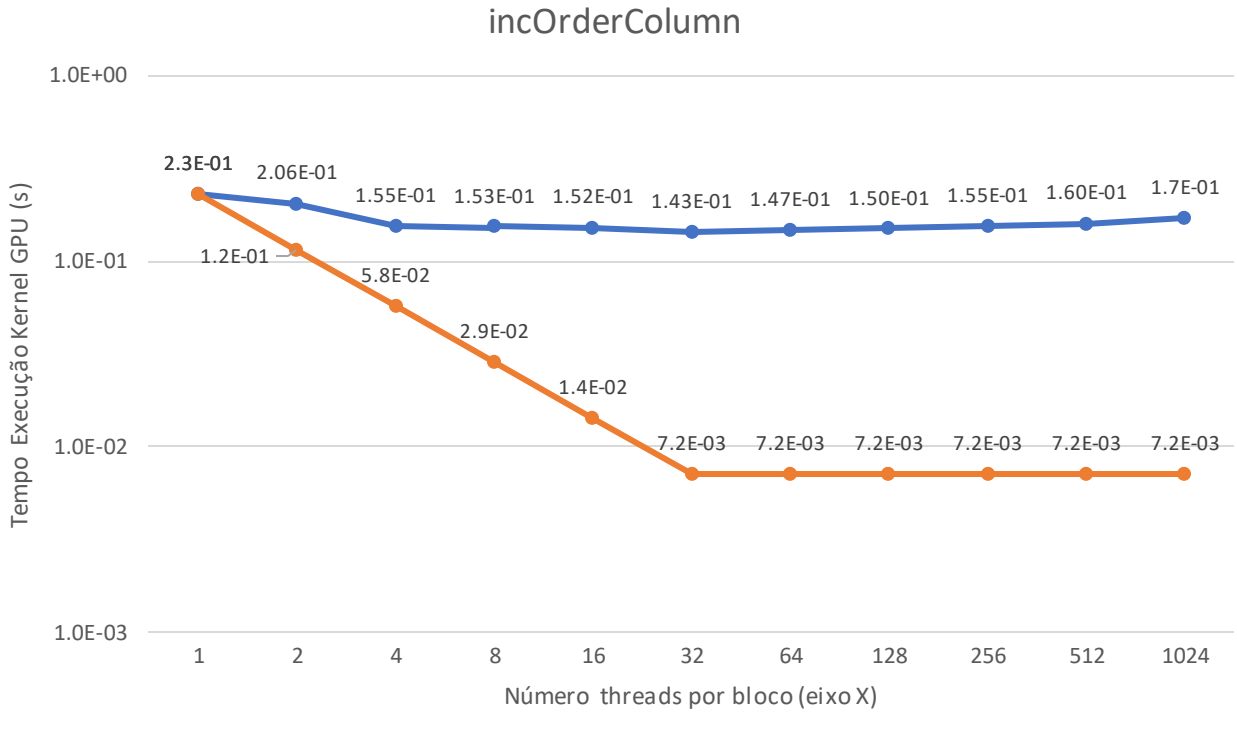


incOrderColumn - analysis

As a first approach we varied the number of threads per block on the x axis and created the following graph for analysis. We can observe that the real execution time doesn't match up with the theoretical expectation. Overall, there is little variation in execution time with the increase of the number of threads per block. The biggest execution gains exists between 2 and 4 threads per block.
The best execution time was obtained with 32 threads per block which matches up with the Warp Size of 32 of the GPU used. This value is consistent with our initial theory, given that multiples of warp size allow for full warp occupancy and therefore max performance.

Although adjacent threads now access contiguous memory regions, every element of each sequence is now very far from each other, so every time a thread accesses a new sequence element there is a high likelihood of that element not being present in the cache.

As the number of threads per block reaches 64 (which results in 16 blocks) and up, the number of blocks is no longer enough for full GPU occupancy given that there are 24 SMs in the GPU used. Even with this factor in play, the execution time suffered very little, because every sequence access requires the values to be fetched from main memory allowing for the SM to run another warp while waiting for the memory transaction to complete thus hiding the memory access latency. This is the main reason for the lack of scalability of the execution time.



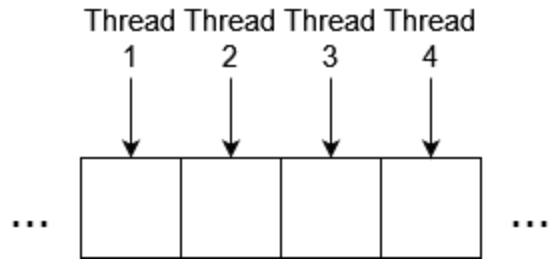


Figure 4: Memory access scheme

incOrderColumn - analysis

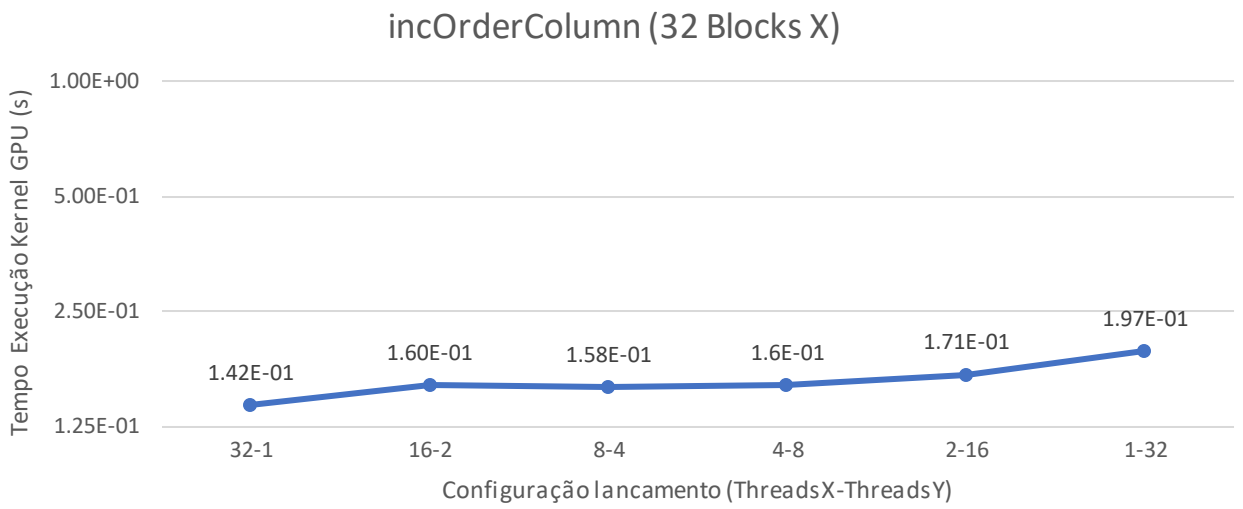
After determining that 32 threads per block offered the best performance value, we varied the block launch configuration on the X and Y axis. The original value of 32 threads per block remained the best performing one since now subsequent threads no longer access contiguous memory resulting in longer execution times.

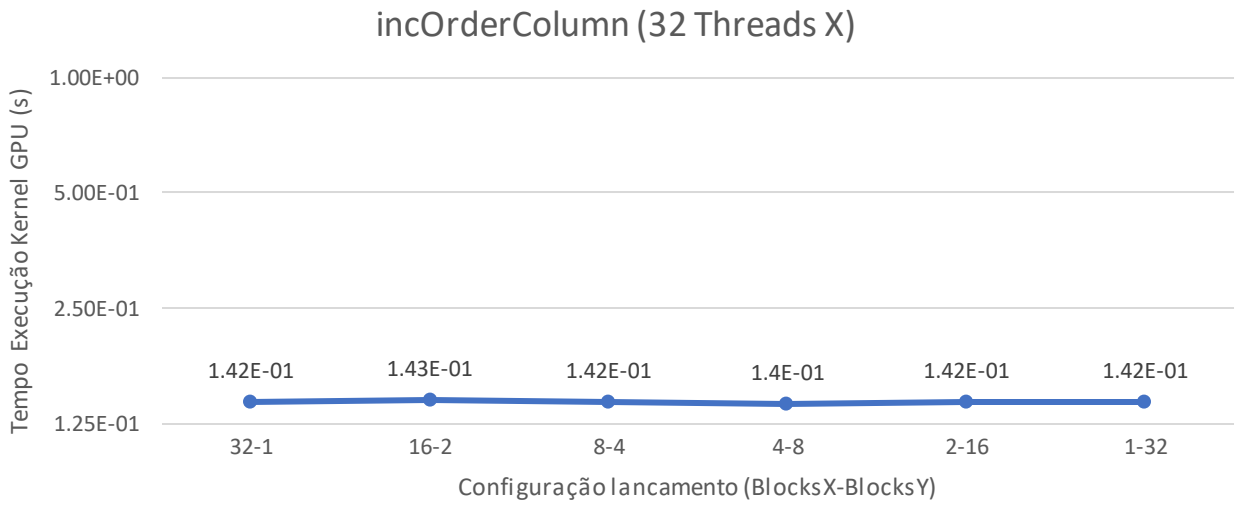
We then varied the Grid Launch Configuration on the X and Y axis maintaining the best Block configuration. The performance variation we observed was very minimal, although best results were obtained for a 4 x 8 Grid. Since every thread in a block is scheduled to the same SM, a variation of grid dimensions resulting in the same Block count should offer little to no performance advantage which coincides with what we observed.

Best Configuration: Grid of 4 * 8 Blocks of 32 threads each

Average CPU time: 8.9E+0 seconds

Conclusion: Although the mapping appears better initially, the lack of the principle of locality ends up hurting the program so it does not scale as expected with increased levels of parallelism. The CPU time in this program is much higher than the GPU version given that the CPU is sorting the sequences sequentially and every element of each sequence is not contiguous resulting in very high cache miss rates.

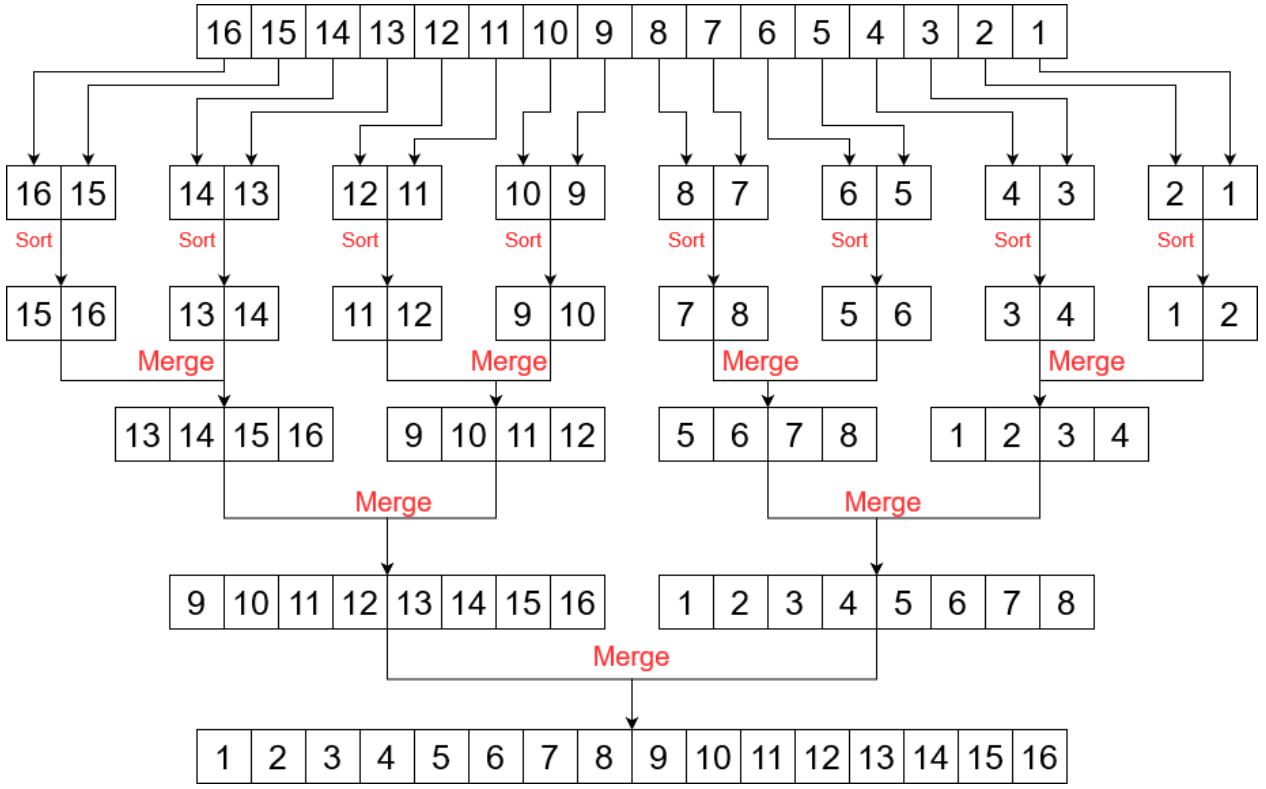




Sorting big sequences

In order to solve this problem while taking advantage of parallel processing power of a GPU we thought a divide and conquer approach would be the best path.

- Assume we have a big sequence of N elements, and we divide the sequence in $M = \frac{N}{S}$ sequences where S is the number of elements of each sub-sequence.
- These sub-sequences can be sorted the same way they were in the original programs.
- After the sorting is complete, we have M sorted sequences of S values.
- We now need to merge this sorted sequences together to get the sorted complete sequence. By merging the M sequences of S elements in pairs we can perform $\frac{M}{2}$ merging operations in parallel to obtain $\frac{M}{2}$ sequences of $2 * S$ elements. This process can be done repeatedly until the complete sorted sequence is obtained.

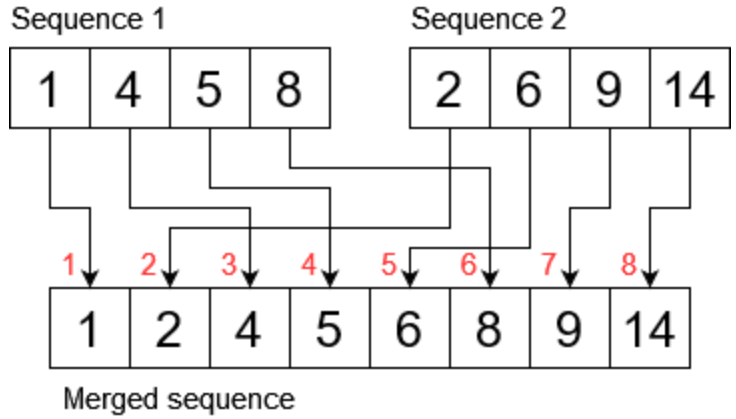


It should be noted, however, that every merging loop the parallelism level will be reduced by a factor of 2 and consequently the last iteration will be fully sequential.

Sorting big sequences - 2

Pseudo Code:

- Allocation and data initialization
- Divide the sequence into $\mathrm{M} = \frac{N}{S}$ sequences
- Sort the M sequences in parallel as before with Bubble Sort
- Let H=M/2
- while H>1
  - Merge H * 2 sequences into H sequences
  - H = H / 2



Assuming we intend to sort a sequence of 1 Million elements, a naïve and sequential implementation would require approximately 10^{12} operations, assuming the time complexity of Bubble Sort is $O(n^2)$.

Assuming the merge operation can be done with $O(2N)$ time complexity, with N being the length of the sequences being merged, S being the size of the sub-sequence and M being N/S, we can calculate the number of required operations with the following expression: $S^2 * M + \log_2 M * N$.

Given the same sequence of 1 Million elements as before, split as 1024 sub-sequences of 1024 elements each, our improved version would require $1.01 * 10^9$ operations, most of which are run in parallel.