0.1 Analysis of work partitioning granularity in the Executors

Dividing the work at thread-level the traditional way, i.e. in as many partitions as processing units we have available, has a drawback: if, for instance, some of the partitions finish earlier than others, the processing units will remain in idle state waiting for the remaining ones to finish, leading to a suboptimal exploitation of the hardware resources. However, we can mitigate the time workers spend in idle state due to an unbalanced distribution of the workload by increasing the granularity of the data partitions.

In this Section, we analyze the impact that the number of partitions we divide the workload into has on the speed up obtained when parallelizing a job with ROOT's executors. Our analysis is executed over different sizes of data and an increasing number of processing units, for both TProcessExecutor and TThreadExecutor. In each case, we increase the number of tasks to execute, moving progressively from coarse-grained partitions, e.g. dividing the workload in as many partitions as processing units available, to the most fine-grained division, that is, one task per data element.

For this purpose, we designed a benchmark consisting of the conditional evaluation of two different implementations of the Vavilov probability distribution function—fast and accurate—which provides us with a simple way to create imbalances in the execution time for the chunks. The fast implementation of the Vavilov probability density function [1] is about 5 times faster for the calculation of the distribution function, while its accurate implementation [2] sacrifices speed for correctness. We perform the evaluation of the Vavilov fast function on the positive values of a provided data set, and the Vavilov accurate evaluation on the negative values of the same data set. Listing 1 showcases the code necessary to perform our benchmark with an explicitly chunked MapReduce operation with TThreadExecutor, given a container of data, `data`, and a number of chunks, `nChunks`.

This benchmark is executed on a Intel Core i7-4790 desktop server with 4 cores at 3.6 GHz and 8 GB of RAM, each core supporting hyperthreading, over two different data sets, both of them generating unbalanced executions. The first data set is generated randomly following a Gaussian distribution with $\sigma = 1$ and $\mu = -0.25$ for the first third of data elements, $\sigma = 1$ and $\mu = 0.25$, for the second third, and $\sigma = 1$ and $\mu = -0.75$ for the remaining third.

Figure 1 displays the speed up results obtained for different sizes of the data set when parallelizing with a different number of threads (including hyperthreading) in TThreadExecutor, increasing the number of data partitions in each case. In Figure 1a we can observe that, for 100 points, the execution time is dominated by the task overhead, causing a slowdown in the total execution time, independently of the

```
// (... variable initialization: data, nChunks)

ROOT::Math::VavilovAccurate vavilovAc(0.3, 0.5);
ROOT::Math::VavilovFast vavilovFast(0.3, 0.5);

auto mapFunction = [&data, &vavilovAc, &vavilovFast](const unsigned i){
   return data[i]>0? vAccurate.Pdf(data[i]) : vFast.Pdf(data[i]);
};

auto redFunction = [](const std::vector<double> vec){
   return std::accumulate(vec.begin(), vec.end(), .0);
};

ROOT::TThreadExecutor  threadPool(nThreads);
auto partialRes = threadPool.Map(mapFunction, ROOT::TSeqU(data.size()), redFunction, nChunks);
```

Listing 1: Example code of our benchmark, computing the Vavilov probability density function in its fast implementation for negative values of the data, and its accurate implementation for positive values of the same data.

number of partitions of the data. Results improve as we increase the dataset size until stabilizing, as shown in Figure 1b. Here we observe significant results in speed up, which improves slightly until, again, task overhead begins to negatively affect the execution time when dealing with smaller tasks.

This example exhibits its best results when relying on hyperthreading. With hyperthreading we observe a gradual, more pronounced, improvement in speed up, reaching up to a remarkable 45% better speed up than the non-hyperthreaded case. However it exhibits a more accentuated decline in speed up when the task creation overhead starts to be noticeable.

Figure 2 shows the results obtained for a multiprocess execution with TProcessExecutor of the same example. While multiprocess execution is more useful in several contexts (e.g. when performing non thread-safe operations), and results display a good speed up for big data sets, the task overhead introduced by TProcessExecutor is several orders of magnitude greater than that of TThreadExecutor, requiring a greater amount of work for the speed up to be noticeable. For this same reason, the plots in Figure 2 exhibit earlier decline of speed up with a higher number of tasks.

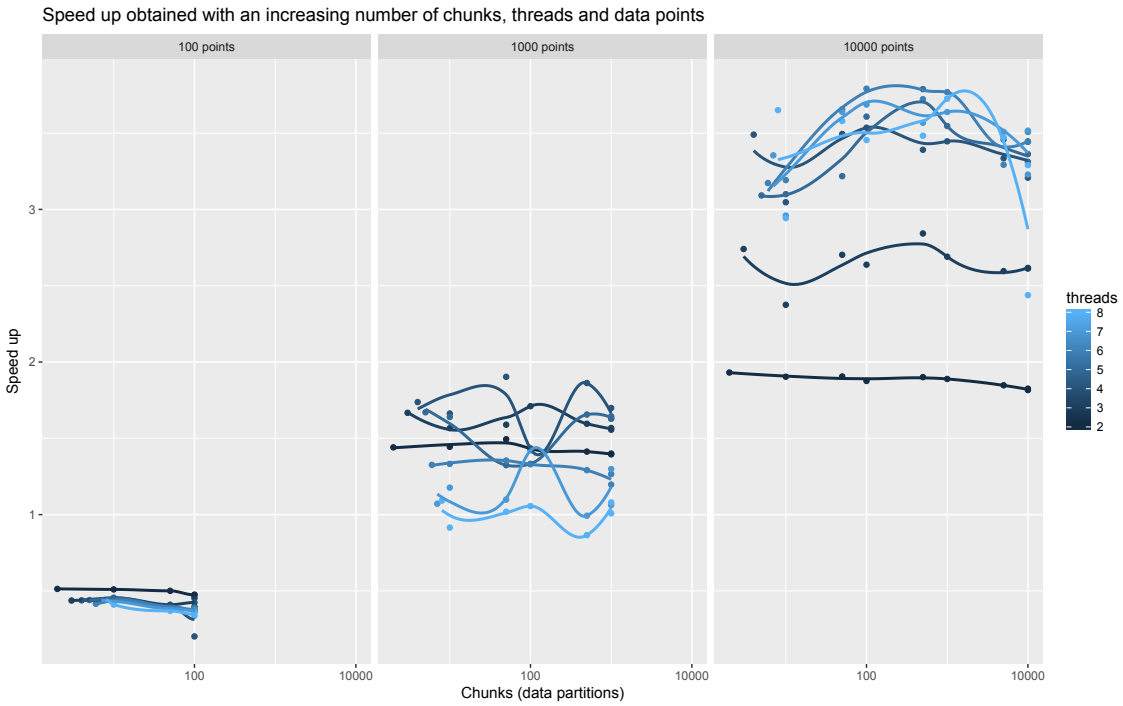
The near-optimal speed up we obtain in the theoretically most unbalanced case (dividing the workload into as many tasks as processing units), might indicate that the data set may not be unbalanced enough to observe dramatic improvements by tuning task granularity. We observe this situation in Figure 3, produced by Intel's Vtune Amplifier, where we visualize the performance trace of an execution of our benchmark with 800 million points, 4 processing units and 4 tasks for both TProcessExecutor and TThreadExecutor. Figure 4 shows the trace of an execution with the same parameters but with an increase in the number of tasks to 8000,

demonstrating improved workload balancing.

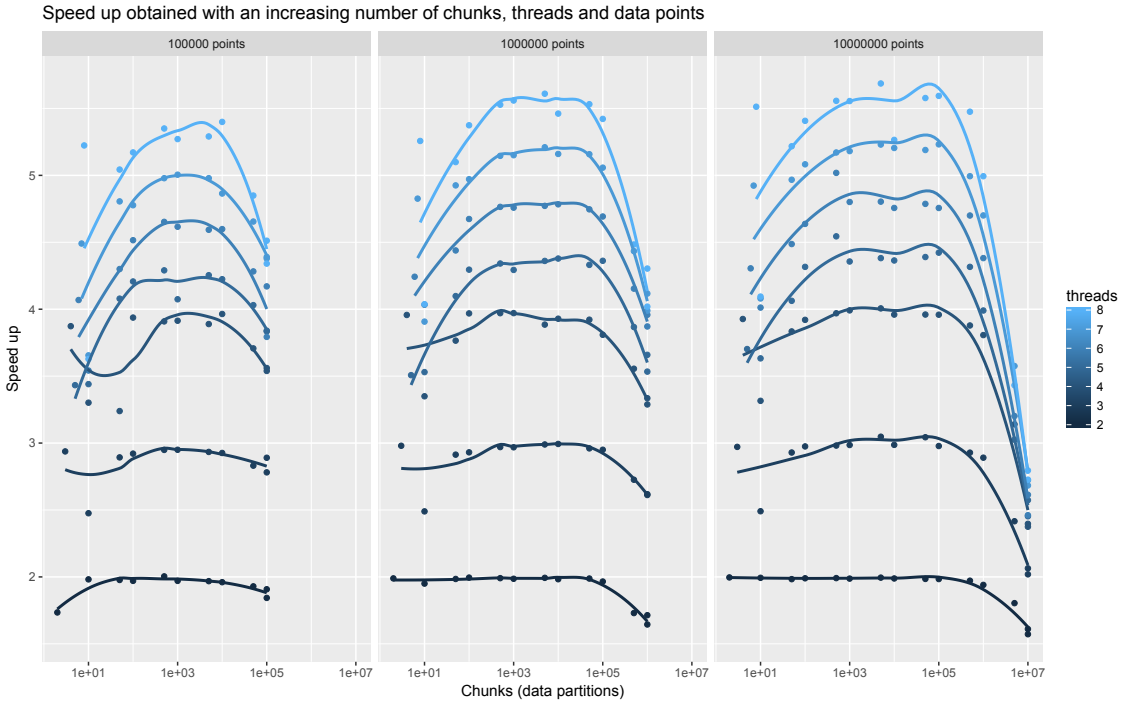
The second data set, which is more extremely unbalanced, is composed of two differentiated parts: the first two thirds of the data set evaluate the fast version of the Vavilov distribution and the remaining third performs the accurate evaluation. Figure 5 demonstrates that this case exhibits a more accentuated imbalance between the processing units. The orange trace in Figure 5a represents the spin and overhead time of the main thread, after finishing its assigned tasks and while it waits for the remaining threads to finish their execution. Again, we provide in Figure 6 a better balanced example of the same execution, achieved by increasing the number of tasks to 8000.

Figure 7 and Figure 8 present the results of executing our benchmark with this second data set. They display, for the multithreaded and the multiprocessed cases, respectively, the increased speed up we obtain when varying task granularity in different pair configurations of number of threads and size of the workload. In this case, we observe an earlier steeper increase in speed up for the higher values of the chunk size, indicating the convenience of splitting tasks into smaller partitions. In the last configuration of Figure 7b, we perceive a gradual decrease in speed up caused by excessive fine-graining of the task, whose execution time becomes dominated by the task overhead. Figure 8 exhibits, again, an earlier decay of the speed up due to the higher task overhead inherent to a multiprocess solution.

These results support the importance of an adequate task size and showcase the impact it might have on performance for unbalanced parallel workloads. Moreover, they indicate that granularity has to be treated differently for the multithreaded and the multiprocessed cases, keeping task creation overhead in mind.



(a) Speed up for or 10^2, 10^3 and 10^4 data points.



(b) Speed up for or 10^5, 10^6 and 10^7 data points.

Figure 1: Speed up obtained with TThreadExecutor for different number of data points generated for the first benchmarking data set, divided into an increasing number of chunks. The measures have been taken with different number of threads.

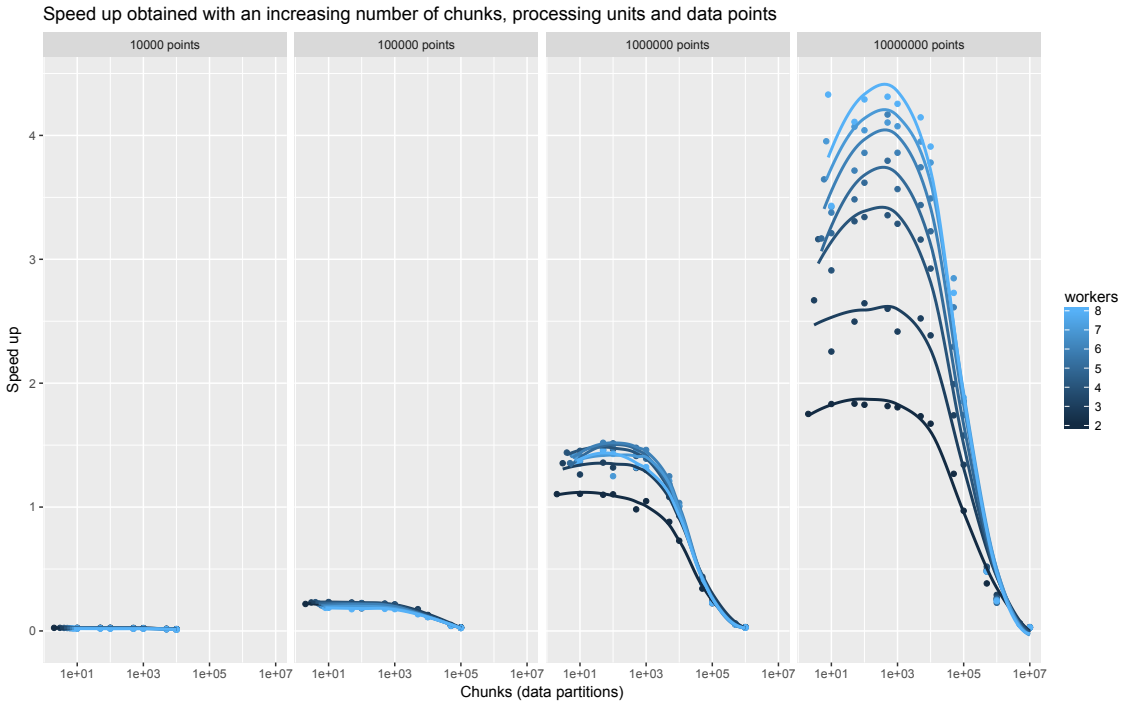
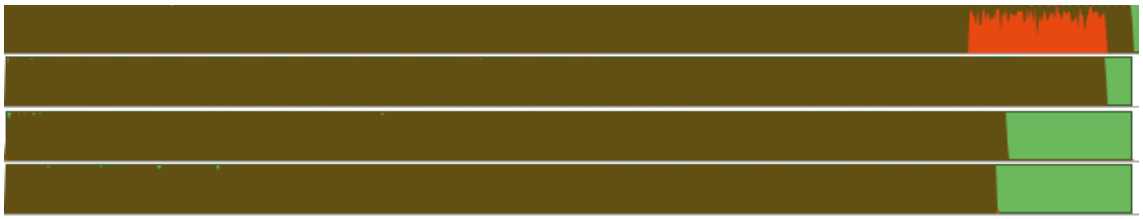
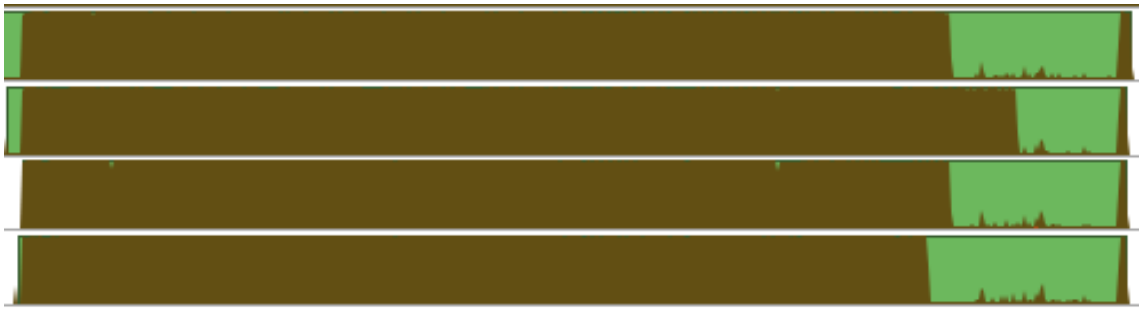


Figure 2: Speed up obtained with TProcessExecutor for different sizes of the data and different number of chunks when benchmarking the first dataset. The measures have been taken with different number of processing units



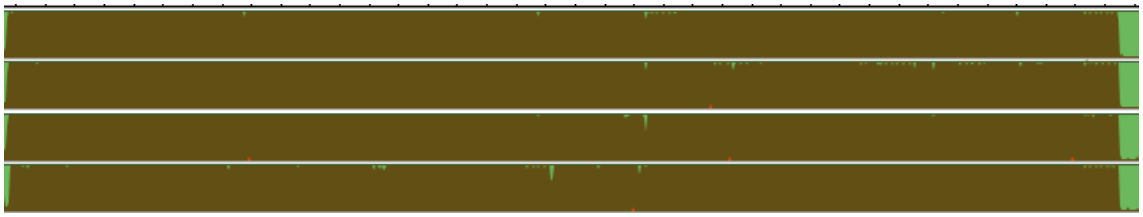
(a) Multithread execution with TThreadExecutor.



(b) Multiprocess execution with TProcessExecutor.

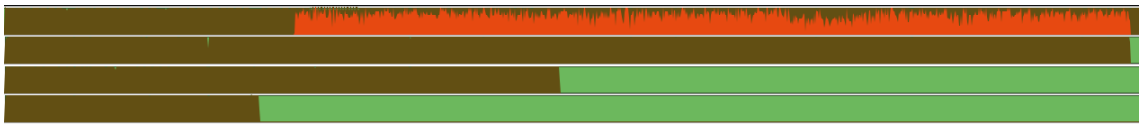
Figure 3: Thread monitoring for sample size of $8 * 10^7$ points generated for the first data set, divided into 4 partitions ($2 * 10^7$ points per chunk) and evaluated by 4 threads.

(a) Multithread execution with TThreadExecutor.

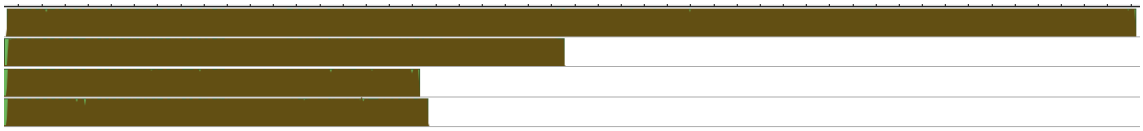


(b) Multiprocess execution with TProcessExecutor.

Figure 4: Thread monitoring for a sample size of $8 * 10^7$ points generated for the first data set, divided into 8000 partitions (10^4 points per chunk) and evaluated by 4 threads.



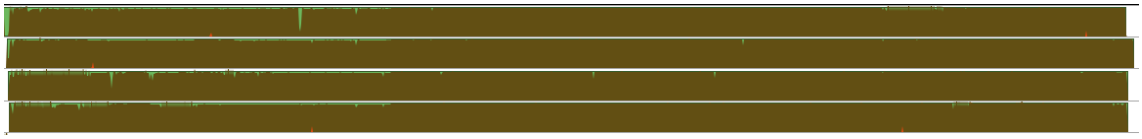
(a) Multithread execution with TThreadExecutor



(b) Multiprocess execution with TProcessExecutor.

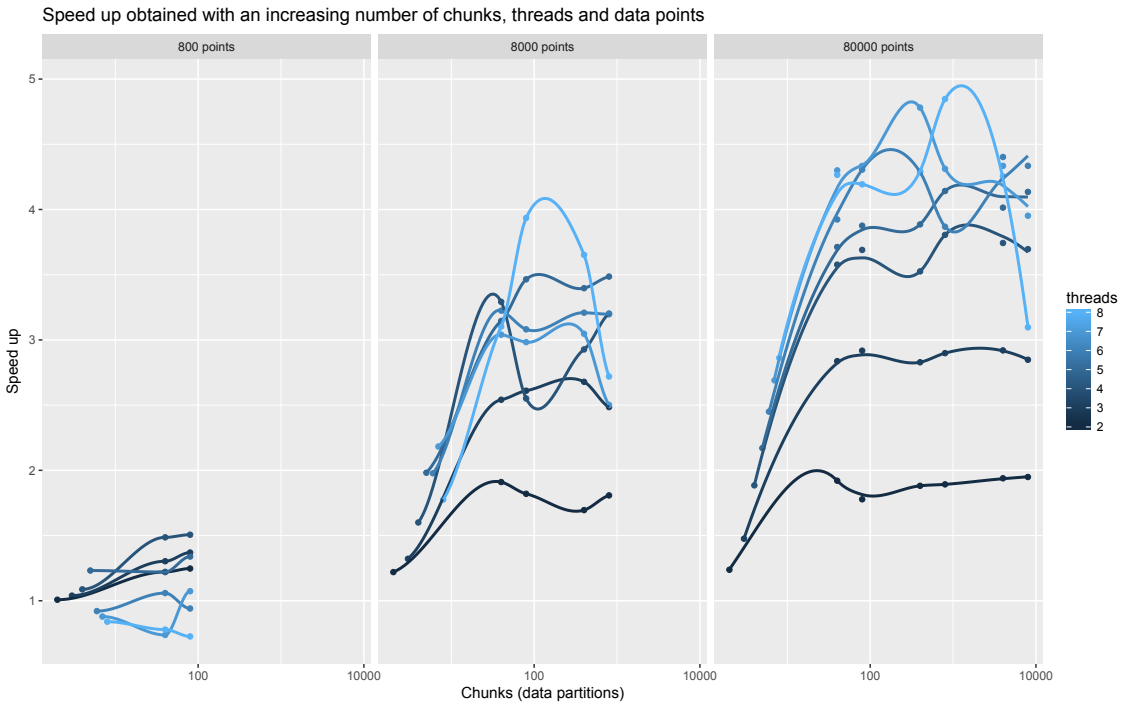
Figure 5: Thread monitoring for sample size of $8 * 10^7$ points generated for the second data set, divided into 4 partitions ($2 * 10^7$ points per chunk) and evaluated by 4 threads.

(a) Multithread execution with TThreadExecutor.

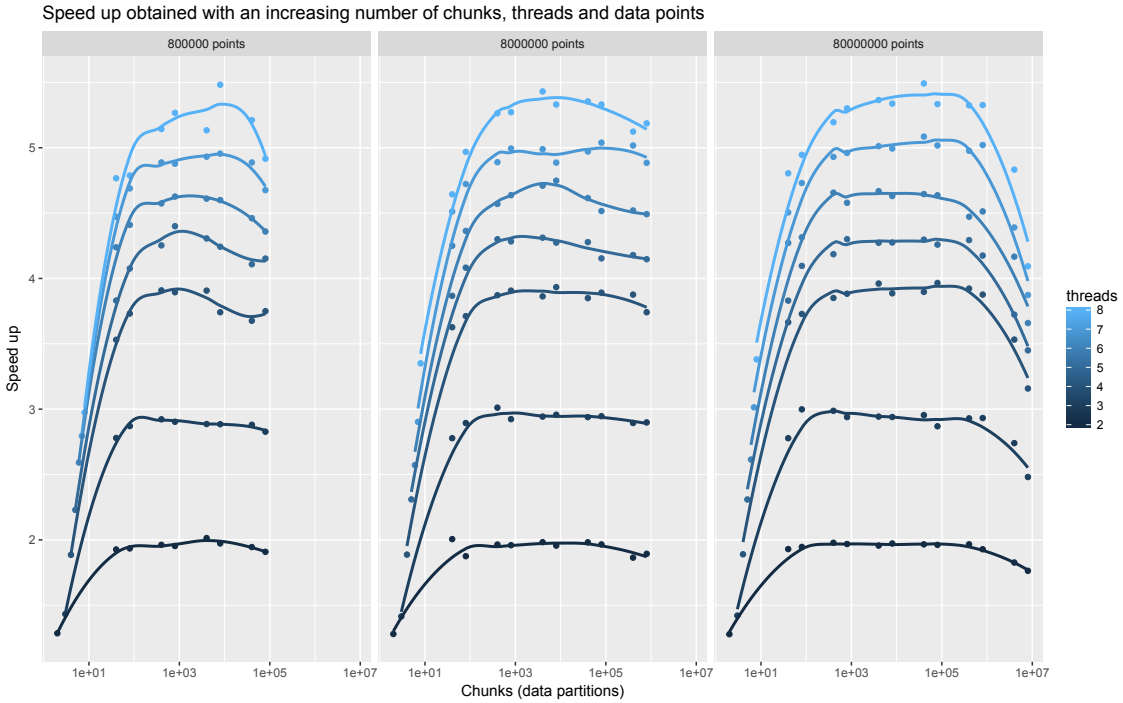


(b) Multiprocess execution with TProcessExecutor.

Figure 6: Thread monitoring for a sample size of $8 * 10^7$ points generated for the second data set, divided into 8000 partitions (10^4 points per chunk) and evaluated by 4 threads.



(a) Speed up for 10^2, 10^3 and 10^4 data points.



(b) Speed up for or 10^5, 10^6 and 10^7 data points.

Figure 7: Speed up obtained with TThreadExecutor for different number of data points of the second benchmarking dataset and an increasing number of chunks. The measures have been taken with different number of threads.

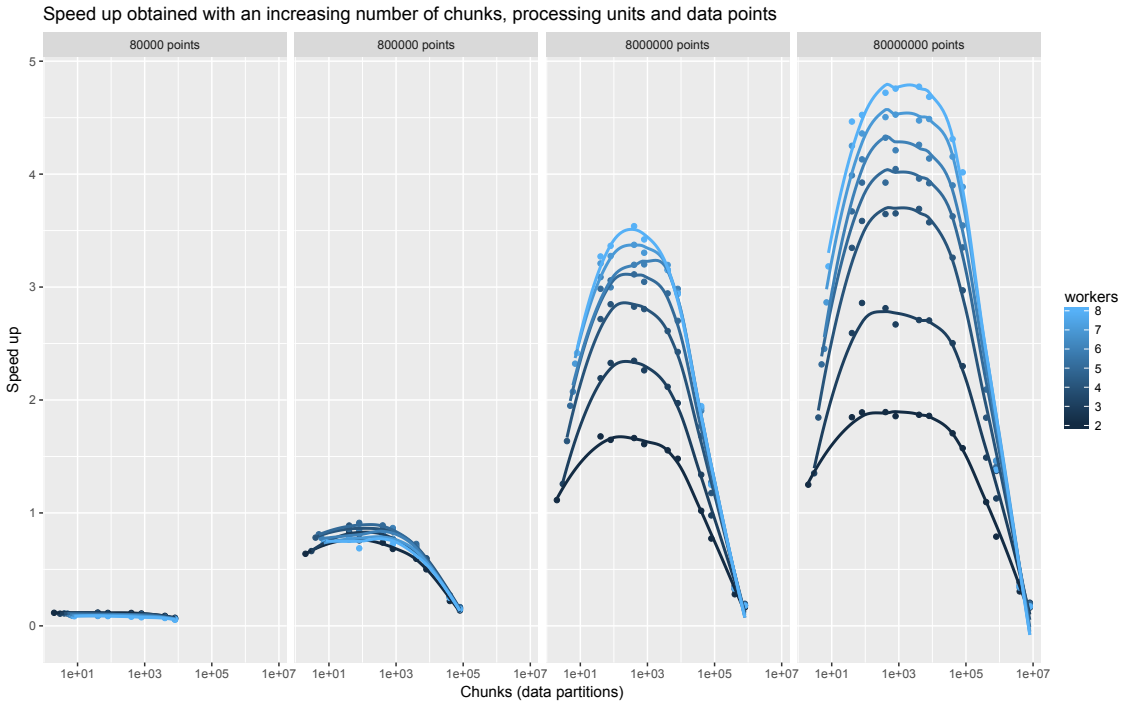


Figure 8: Speed up obtained with TProcessExecutor for different sizes of the data and different number of chunks when benchmarking with the second dataset. The measures have been taken with different number of processing units

Bibliography

[1] Alberto Rotondi and Paolo Montagna. Fast calculation of vavilov distribution. *Nuclear Instruments and Methods in Physics Research Section B: Beam Interactions with Materials and Atoms*, 47(3):215–223, 1990.

[2] B Schorr. Programs for the landau and the vavilov distributions and the corresponding random numbers. *Comput. Phys. Commun.*, 7(CERN-DD-73-26):215–24, 1973.