Closest Pair of Points - MPI Parallel implementation

Davide Modolo (#229297), Kevin Depedri (#229358)

Abstract
The goal of this project is to develop a *parallel implementation* to find the minimum distance between two points across a set of n points in k dimensions. To achieve this result we implemented four different approaches, the first two are sequential (Bruteforce and Divide et Impera), while the second two are their parallel versions, implemented using MPI. Furthermore, to achieve greater performances, also the Merge Sort algorithm has been fully parallelized, since it is a fundamental component of the Divide et Impera approach. Once developed, the parallel algorithms have been deployed using the University of Trento HPC-cluster, with different configurations of input points, processors and nodes, which allowed to assess their performances under various conditions. All the results and the implementations can be found on our GitHub repository[1].

Introduction

The Closest Pair of Points problem is a classic computational geometry problem. It involves finding the two points in a given set of points that are closest to each other in terms of their Euclidean distance.

This problem has numerous applications in various fields, such as computer graphics, pattern recognition and geographic information systems, and it can be addressed in different ways.

The time complexity of these methods can vary, ranging from $O(n^2)$ for the Bruteforce implementation, where each point is compared with all the other points, to $O(n \log n)$ for the Divide et Impera implementation, which has been proven to be the most efficient approach to solve this problem.

1. Task formalization

The Closest Pair of Points problem can be formalized as follows:

Task: Find the closest pair of points in the input set in terms of their Euclidean distance.

Input: A set of n points in a k-dimensional space, where each point is represented as a k-dimensional vector. We represented points using a C struct, where each point defines its number of dimensions and its coordinates.

Output: The pair of points in the input set that are closest to each other[1], along with their Euclidean distance.

Formally, the **Euclidean distance** between two points p_a and p_b is defined as the square root of the sum, along k dimensions, of the square difference between the corresponding coordinates of the points in that dimension:

$\text{distance}(p_a, p_b) = \sqrt{\sum_{i=0}^{k}(a_i - b_i)^2}$, where p_a and p_b are two points in a k-dimensional space, and $a_1, a_2, ..., a_k$ and $b_1, b_2, ..., b_k$ are the coordinates of the points.

[1] alternatively, if more than one pair is present, all the pairs that have the same minimum distance can be returned

Although we found that most online implementations addressed the problem in 2 dimensions, we chose to adhere to the general problem definition and develop an algorithm that works for any number k of dimensions.

Assumptions Our implementations to solve this problem are based on the following assumptions:

- the minimum distance between two point in the space is less than INT_MAX;[2]
- the maximum number of points in the space is less than (or equal to) INT_MAX;
- there are no duplicate points (otherwise the minimum distance would be 0);
- coordinates are integers (but they can easily be extended to non integers).

Point generator To run the algorithm it is necessary to have an input set of points. Therefore, we created a Python script that, given a number n of point to generate, a number k of dimensions and the range in which those points will fall, outputs a text file containing the generated points. For large n, we used the cluster computational power to generate such amount of points.

2. Sequential versions

First, we implemented two sequential approaches to solve the problem. Later we translated them in their parallel versions to achieve the results in a shorter time. The first part (reading and saving points from a text file to a list) is shared among the sequential and parallel implementations (more at Section 3).

2.1 Bruteforce

This approach works by checking the distance of a point with respect to every other point of the input set. The time complexity is $O(n^2)$ and, just for a quick comparison with the next

[2] INT_MAX = 2147483647

approach, with a set of 1 million points it took around 7 hours to find the closest pair of points.

2.2 Divide et Impera

This approach works by first sorting the points over one dimension (usually x) and then recursively splitting the space in half until 2 or 3 points are present in each sub-space. After that, all the possible distances between the points present in each sub-space are computed, and, for each split, the smallest distances of its two sub-spaces (d_L and d_R) are obtained.

Then, for each split, only $d = \min(d_L, d_R)$ is kept as minimum distance obtained for that split. This minimum distance d is used to define the width of a **strip**, which is centered on the division line between the two sub-spaces of a split, with a total width of $2d$ (gray area in Figure 1).

Once the strip is defined, the points that fall inside it are ordered according the next dimension of the space (usually y). Then, the distance between these point is computed, starting from the point with the lowest value of y (let's call it p_i) comparing it with points with increasing values of y, until the resulting difference over the coordinate y of the two points is smaller than d. Indeed, as the distance between two point over the y dimension gets bigger than d, we already know that these two points, and all the next possible combination of points with p_i, will not have an Euclidean distance smaller than d, for this reason we can break the computation for the point p_i and repeat the procedure for the next point p_{i+1} in the strip.

This process allows us to recursively merge back all the sub-spaces until we get to the initial one, returning the closest pair of points in $O(n \log n)$ time.

As seen here above, this approach requires sorting the array multiple times (first along x, then for each split also along y). For this reason we implemented the **Merge Sort** algorithm, which allowed us to sort the points in a very efficient way. Indeed, the time required for this approach (relying on merge-sort) to find the closest pair of points in a set of 1 million points is 28 seconds.

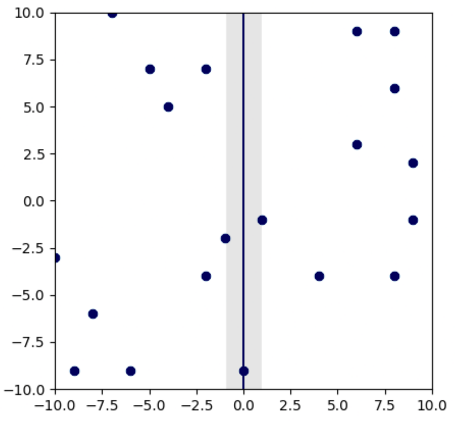

Figure 1. Example of split of the space in two sub-spaces, followed by the computation of the strip of width $2d$ which merges the two sub-spaces back

3. Parallel versions

Once the sequential versions had been fully implemented, we started converting them to parallel using MPI.

To be more specific, the main MPI functions we used in our implementations are:

`MPI_Allreduce` with `MPI_Min`, it allows to *gather* the wanted value from all the processes, perform a mathematical operation (in this case *min*) and to *broadcast* the result (in this case the minimum distance found) back to each process

`MPI_BCast`, it allows to send any needed values to all processes (in our case: number of total points, number of local points and number of dimensions)

`MPI_Pack - MPI_UnPack`, they allow to pack data (in our case the struct relative to the points that we want to send) to then transfer it. Once received, we unpack the data and we use it to fill new structs which is allocated in the receiving process

`MPI_Send - MPI_Recv`, they allow to send and receive values (in our case points, out of region values, and other variables) or other type of data (like the packed points) between processes

Our initial implementation of the parallel algorithm tried to perform a partial read of the points on each process to reduce communication overhead. The idea behind this approach was to compute the number of bytes to skip for each process when reading the input file. Unfortunately, we encountered two main problems: inconsistent byte counting and concurrent reading from different processes, which resulted in unreliable behaviors. Therefore, the implementation was changed to have only the master process (process 0) read the input file and send the exact number of points to each available process.

Another improvement that we introduced in the implementations regards the output obtained. Indeed, this type of problem normally assumes that the 'minimum distance is returned, together with the two points that led to that distance'. We know that, as the number of points present in a bounded space increase, also the probability of finding multiple pairs with the same distance increases. For this reason we extended the algorithm, introducing some functions to enumerate and to print all the pairs that share the minimum distance. This functionality can be recalled just enabling the corresponding flag when lunching the program.

3.1 Bruteforce

Even if this approach is not the best to solve this problem, we wanted to implement it just to have an additional comparison for the main approach.

The flow can be summarized in the next few steps:

1. the master process reads all the points from the text file

2. the master process sends all the points to every other process;
3. each process compares `num_points`$/($`comm_size`$-1) \cdot$ `rank` points with all the points that it received initially;
4. the master process works with the `num_points`$\%($`comm_size`$-1)$ last points;
5. the smallest distance found is shared among all the processes;
6. the processes with the smallest distance print the pairs used to achieve it.

More in details, the number of points is sent with a call to `MPI_Bcast`, the points are sent using `MPI_Pack/Unpack` with `MPI_Send/Recv` and the smallest distance is obtained and shared with a call to `MPI_Allreduce + MPI_Min`.

Contrary to the sequential Bruteforce implementation, the time required to find the closest pair of points with this parallel implementation is much shorter. Indeed, working with 1 million points we have been able to reduce the time required from about 7 hours (sequential) to about 14 minutes (with 80 cores). This is a good improvement, even though the performances are still much lower than the one of the Divide et Impera algorithm (sequential).

3.2 Merge sort

To perform the Divide et Impera approach, we needed the Merge Sort algorithm. For this reason we decided to parallelize it to achieve greater performances. We will discuss this parallel implementation only briefly since it is an entire different project and it is not the object of our report. The idea behind it is to divide the work of reordering the points along a dimension over different processes as follows:

1. the master process reads all the points from the text file;
2. it sends slices of points to the other processes, keeping for itself the last `num_points`$\%($`comm_size`$-1)$ points. In this way it will never be more loaded than the other processes;
3. every process performs the Merge Sort on its slice of points and sends them back to the master process;
4. the master process performs the merge function to get the final sorted list of points.

The flow of our implementation can be seen in Figure 2.

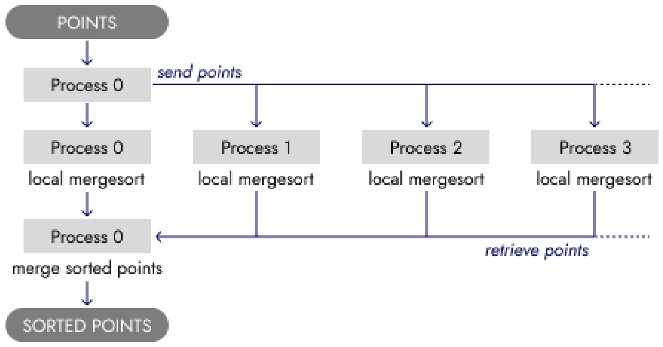

Figure 2. Communication between processes in our parallel Merge Sort implementation

3.3 Divide et Impera

The parallel implementation of the Closest Pair of Points Divide et Impera approach starts with the master process reading the points from the input file and sorting them using the Parallel Merge Sort algorithm discussed before. The sorted list of points is then divided equally among all the processes, with the master process keeping any left-over points.

In addition to a slice of points, every process receives also the x value of the last point of its previous process and the x value of the first point of its following process. These values are then used to compute the borders in between each process's space. Notice that in our implementation the master process doesn't have any process after itself (as it is the last process) and the process number 1 doesn't have any process before itself (as it is the first process).

Each process then performs the Closest Pair of Points algorithm locally on its assigned points, saving the pairs of points that resulted in the local minimum distance.

The `MPI_Allreduce` function, with `MPI_Min`, is then called to share the local minimum distances found among all the processes and determine a temporary global minimum distance.

All the processes then compute their left and right strips by selecting the points that are less than the temporary global minimum distance away from their borders. During this step, the master process only creates its left strip and the process 1 creates only its right strip.

Now all the processes (other than the process 1) send their left strip to the previous process. At the same time, all the processes (other than the master process) receive the left strip sent by their following process. Their own right strip and the received left are then merged and the last part of the Closest Pair of Points algorithm (from sorting over y, Section 2.2) is run on this combined subsection of the space, potentially finding a new local minimum distance.

This new local distance is once again shared among all the processes to define the final global minimum distance (using another `MPI_Allreduce` call). Then each process checks if its local minimum distance is equal to the global minimum distance. If so, the process prints the pairs of points that resulted in its local minimum distance.

By sharing the temporary global minimum distance, the number of comparisons needed is reduced with respect to using the local minimum distance. This helps to optimize the overall performance of the algorithm.

A summary of this algorithm's flow is shown in Figure 3.

4. Benchmarking

To test the performance of the developed algorithms we decided to use a number of cores in the range from 1 to 80, organized in different configurations (packed, packed exclusive,

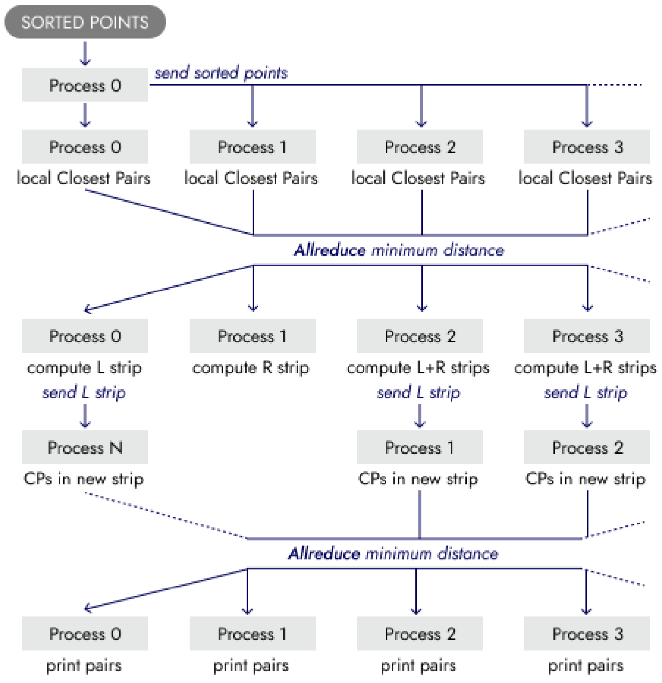

Figure 3. Process of communication in our parallel implementation of the Divide et Impera approach

scatter and scatter exclusive) with many sizes of the input set of points (50M, 100M, 200M and 250M for the Divide et Impera approach, 500K and 1M for the Bruteforce approach). The results obtained for each configuration are shown in Section 6.

4.1 Divide et Impera

In Figure 4, all configurations of the Divide et Impera algorithm exhibit the same trend, with a speedup that increases as the number of cores increases. However, the slope of the curves is steeper at lower core counts, and it gradually decreases as the number of cores increases. Furthermore, on average, higher number of points reduces the speedup.

As a result, the efficiency of the algorithm decreases as the number of cores increases, as illustrated in Figure 5. The declining efficiency is a clear sign that the algorithm is not fully parallelizable and that communication overhead is a limiting factor in its performance.

In Figure 6, it is challenging to identify a specific configuration that outperforms the others. The results depend on the number of points in the input set. It is interesting to note that, looking at Figure 7, the Scatter Exclusive configuration exhibits less stability, with more fluctuations in execution time.

4.2 Bruteforce

Figure 8 shows that the Bruteforce algorithm speedup is almost linear as the number of cores increases, this points out a high degree of parallelization. However, it's worth noting that the efficiency of the Bruteforce algorithm is subjected to an initial drop when the parallelization process starts, as seen in Figure 9. After the initial drop, the efficiency of the Bruteforce algorithm remains quite stable, regardless of the number of cores used.

Interestingly, Figure 10 and Figure 11 show that, independently from the configuration used, the time taken to execute the Bruteforce algorithm decreased in almost the same way.

5. Conclusions

In this report, we described the procedure followed to develop the serial implementation of the algorithm to solve the Closest Pair of Points problem, together with the design choices performed to parallelize them. The results are two parallel algorithms which allows to exploit multi-processing capabilities to find the correct solution of a problem in a shorter time.

More in details, the *Divide et Impera* implementation was able to achieve the best results, since its approach is mathematically superior with respect to the one of the *Bruteforce* implementation. Despite that, the Bruteforce algorithm showed a better parallelization capability, which allowed it to reach greater speedups and efficiency as the number of core increased with respect to the Divide et Impera implementation.

Talking about different node configurations, we observed no significant differences in performance between them, although there were some stronger fluctuations with the *Scatter Exclusive* setting.

Generally speaking, after a certain number of cores, performances reached a plateau. Indeed, the speedup achieved through parallelization may be subject to diminishing returns as the number of cores increases, due to factors such as communication overhead and synchronization.

Nonetheless, the results of this implementation demonstrate the potential benefits of parallelizing the Closest Pair of Points Divide et Impera approach with MPI. Further optimization and tuning could potentially yield even greater performance improvements.

5.1 Future works

- **Improve parallel Merge Sort**, by applying the merge function between each pair of process, instead of merging everything in the master process, to minimize the impact of communication and synchronization.
- **Better partition** the point distribution to assign more work to the master process.
- **Implement concurrent reading** using `fseek` to reduce the time spent by the master process to read and transfer all the input data.

References

[1] GitHub repository with everything we did. [Online]. Available: https://github.com/KevinDepedri/MPI-Parallel-Closest-Pair-of-Points

6. Data

The following pages contain all the plots and tables displaying all the data we acquired.

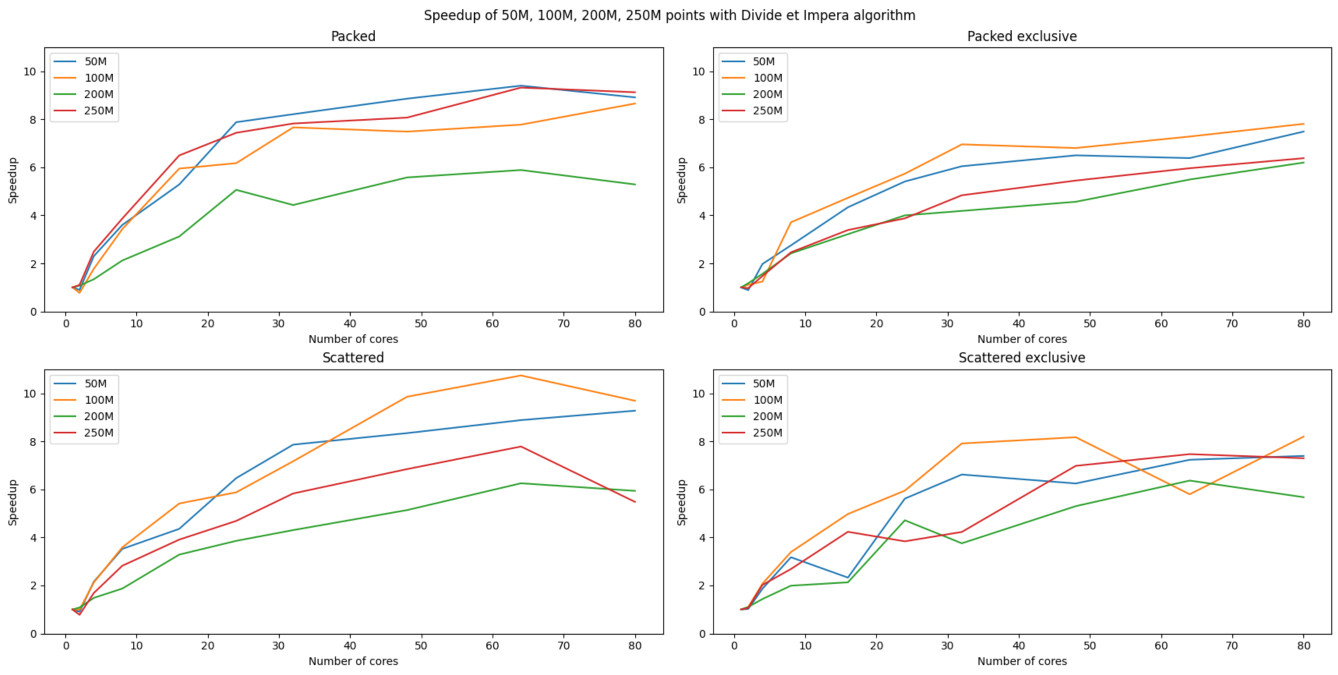

Figure 4. Plots of Speedup over #cores by configuration, Divide et Impera algorithm

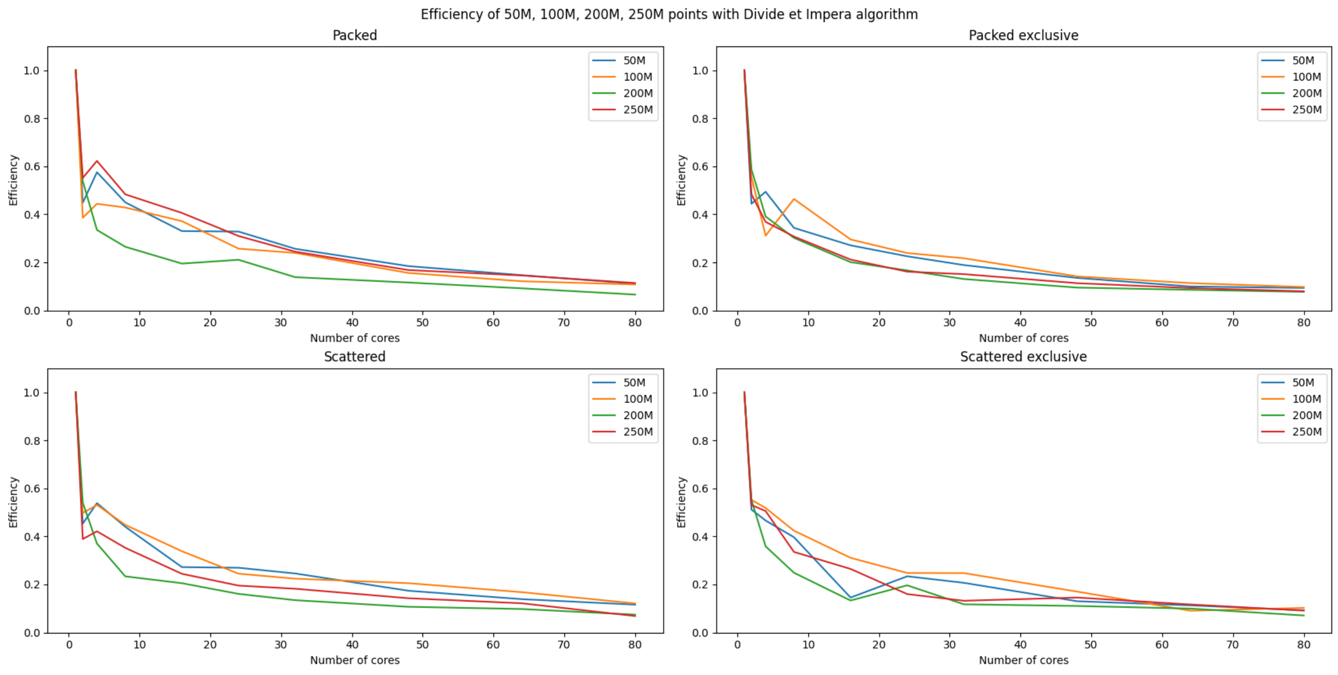

Figure 5. Plots of Efficiency over #cores by configuration, Divide et Impera algorithm

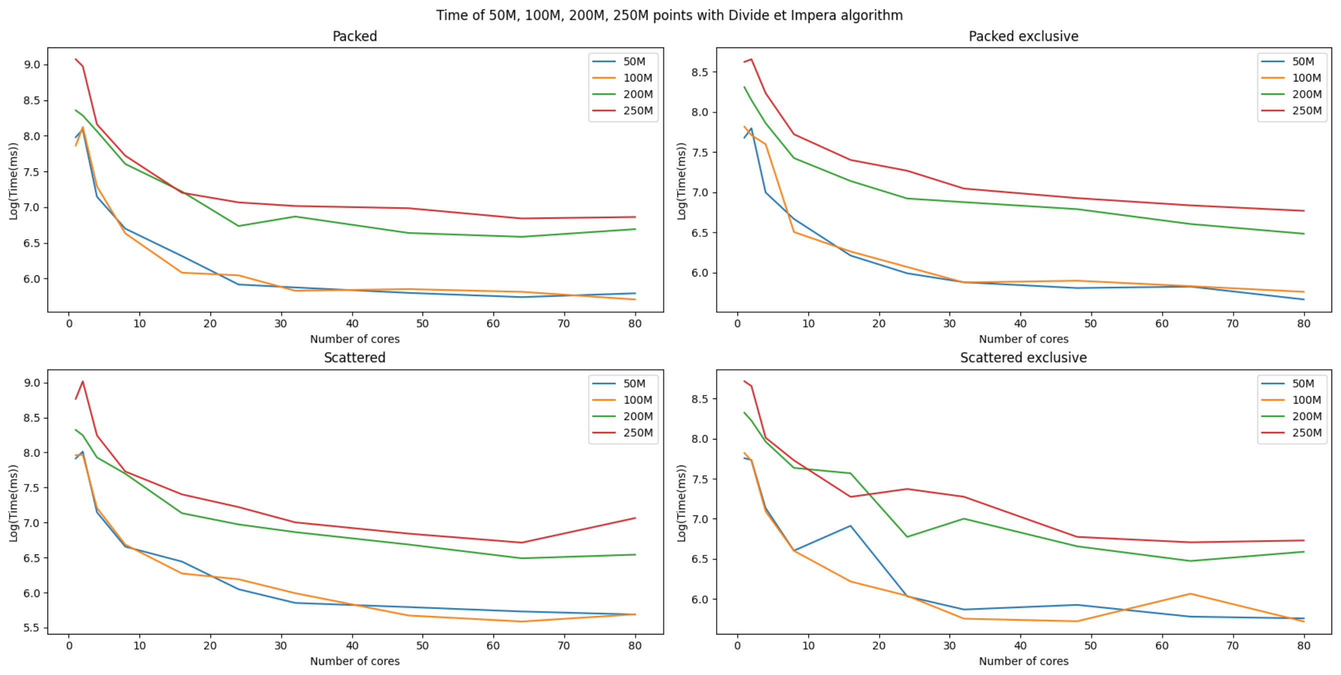

Figure 6. Plots of time over #cores by configuration, log scale, Divide et Impera algorithm

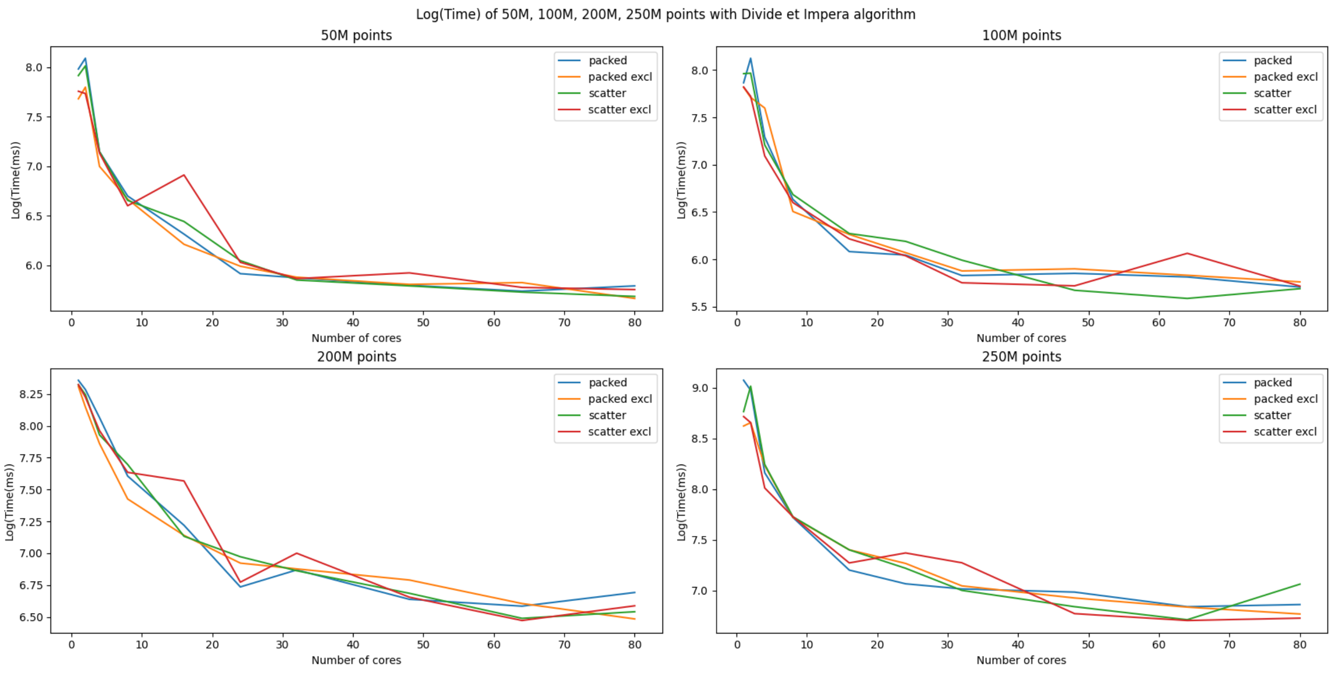

Figure 7. Plots of time over #cores by #points, log scale, Divide et Impera algorithm

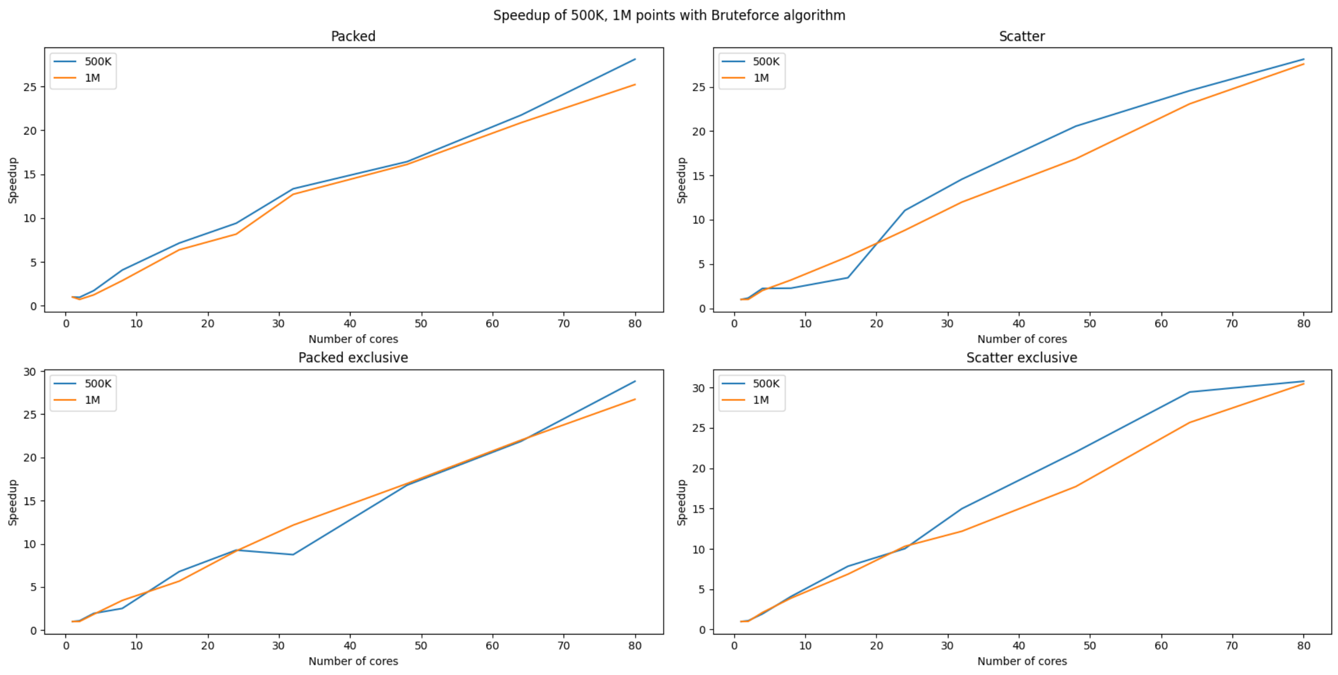

Figure 8. Plots of Speedup over #cores by configuration, Bruteforce algorithm

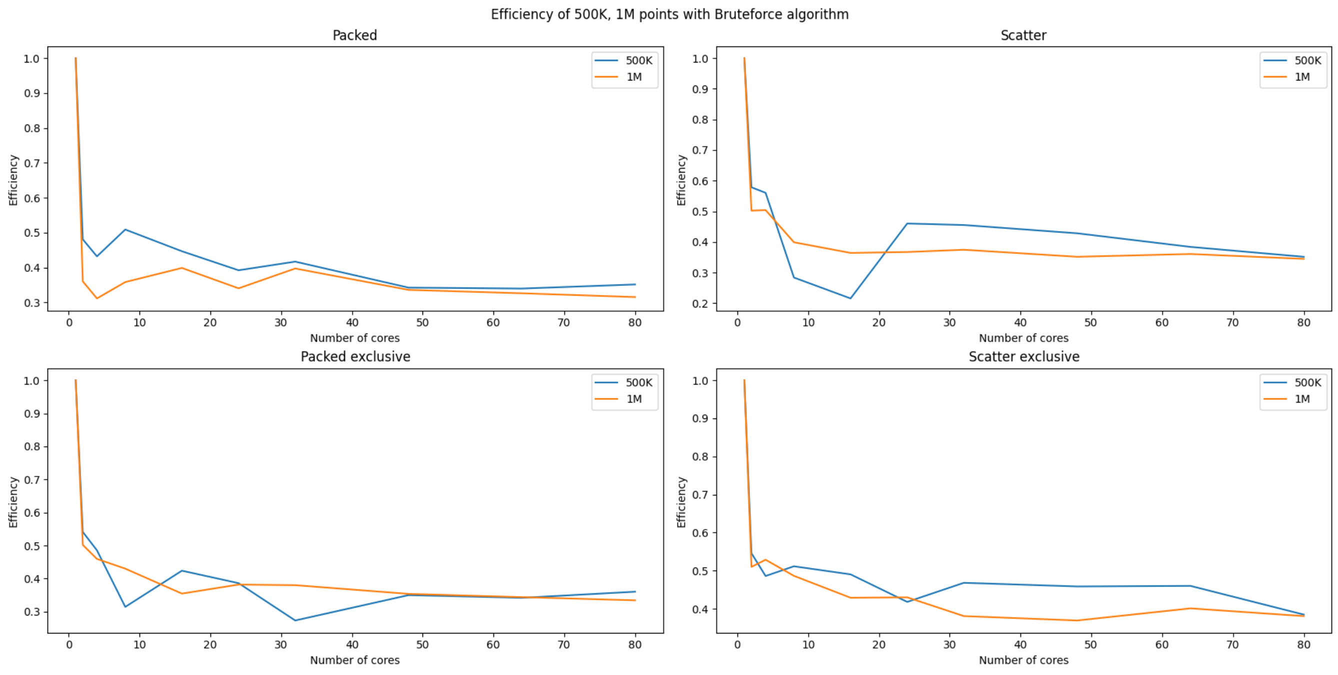

Figure 9. Plots of Efficiency over #cores by configuration, Bruteforce algorithm

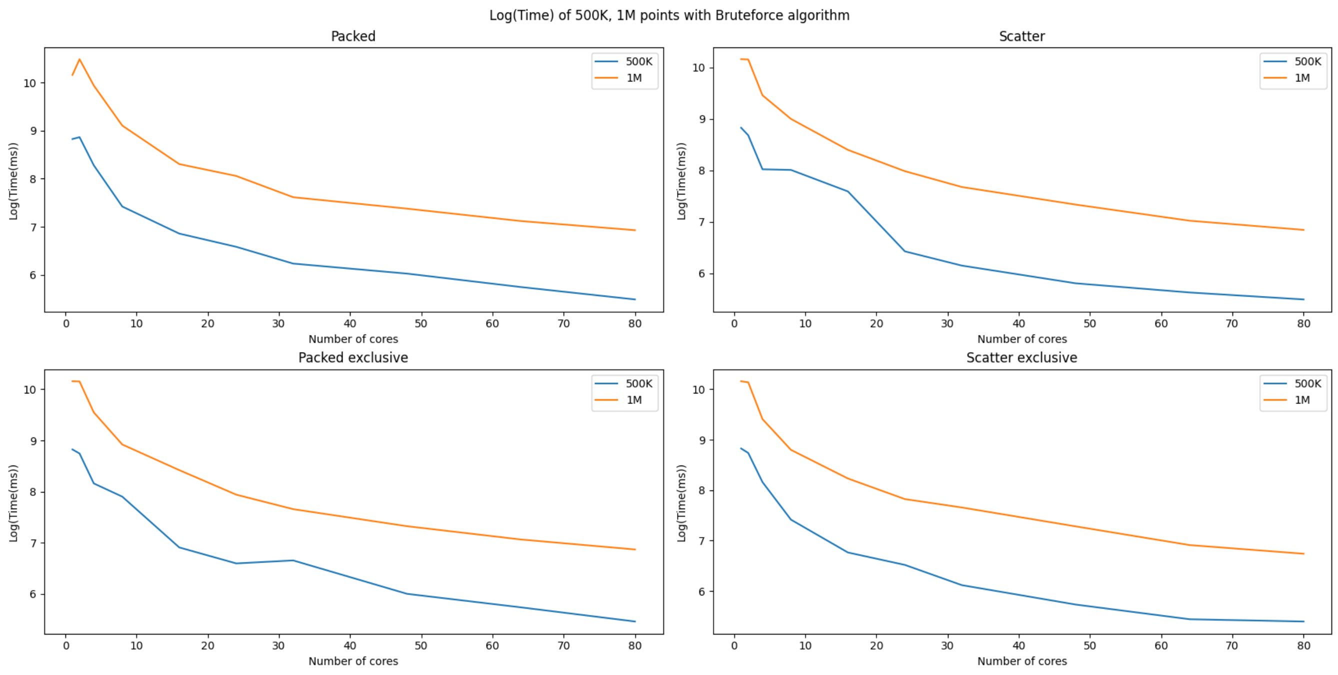

Figure 10. Plots of time over #cores by configuration, log scale, Bruteforce algorithm

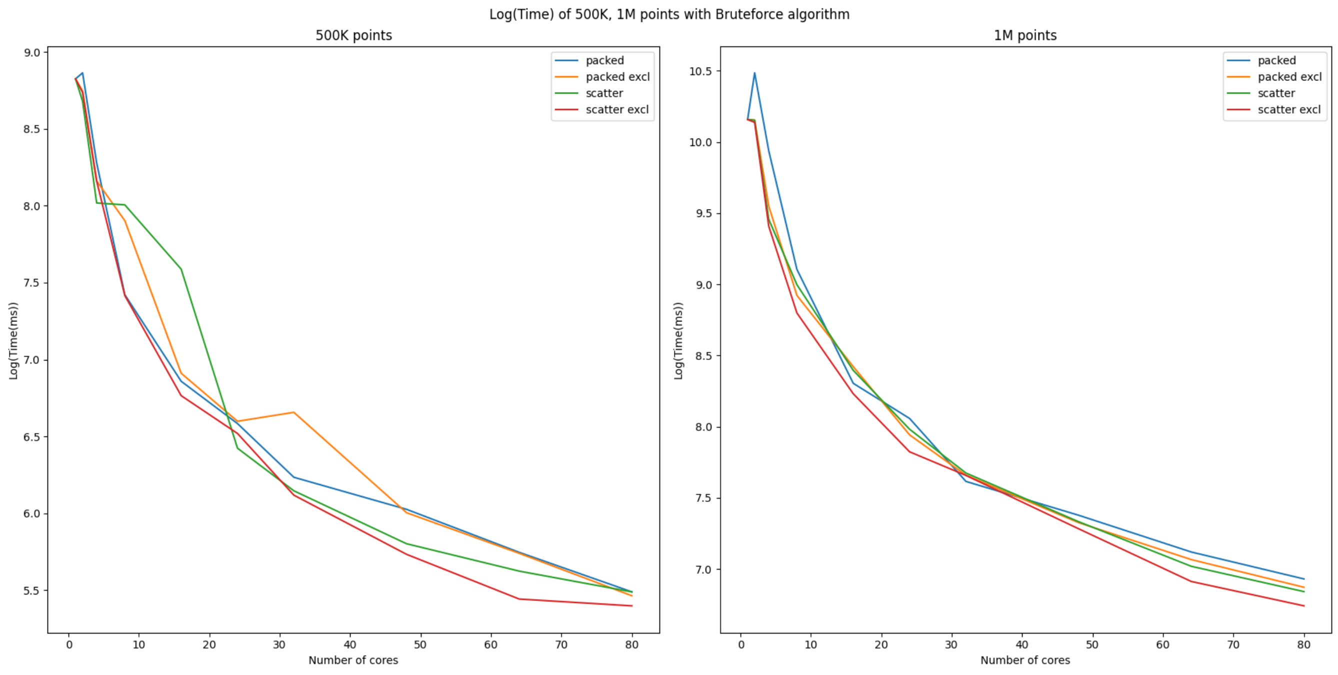

Figure 11. Plots of time over #cores by #points, log scale, Bruteforce algorithm

Table 1. Divide et Impera 50M

Cores	Packed			Packed Excl			Scatter			Scatter Excl		
	Time	Speedup	Efficiency	Time	Speedup	Efficiency	Time	Speedup	Efficiency	Time	Speedup	Efficiency
1	2923	1,00	1,00	2164	1,00	1,00	2736	1,00	1,00	2336	1,00	1,00
2	3260	0,90	0,45	2436	0,89	0,44	3018	0,91	0,45	2282	1,02	0,51
4	1270	2,30	0,58	1095	1,98	0,49	1270	2,15	0,54	1253	1,86	0,47
8	812	3,60	0,45	787	2,75	0,34	777	3,52	0,44	736	3,17	0,40
16	553	5,29	0,33	499	4,34	0,27	628	4,36	0,27	1004	2,33	0,15
24	371	7,88	0,33	400	5,41	0,23	423	6,47	0,27	416	5,62	0,23
32	356	8,21	0,26	358	6,04	0,19	348	7,86	0,25	353	6,62	0,21
48	330	8,86	0,18	333	6,50	0,14	328	8,34	0,17	374	6,25	0,13
64	311	9,40	0,15	339	6,38	0,10	308	8,88	0,14	323	7,23	0,11
80	328	8,91	0,11	289	7,49	0,09	295	9,27	0,12	316	7,39	0,09

Table 2. Divide et Impera 100M

Cores	Packed			Packed Excl			Scatter			Scatter Excl		
	Time	Speedup	Efficiency	Time	Speedup	Efficiency	Time	Speedup	Efficiency	Time	Speedup	Efficiency
1	2605	1,00	1,00	2483	1,00	1,00	2868	1,00	1,00	2492	1,00	1,00
2	3376	0,77	0,39	2231	1,11	0,56	2883	0,99	0,50	2257	1,10	0,55
4	1467	1,78	0,44	1995	1,24	0,31	1350	2,12	0,53	1202	2,07	0,52
8	760	3,43	0,43	669	3,71	0,46	800	3,59	0,45	735	3,39	0,42
16	438	5,95	0,37	525	4,73	0,30	530	5,41	0,34	501	4,97	0,31
24	422	6,17	0,26	433	5,73	0,24	488	5,88	0,24	419	5,95	0,25
32	340	7,66	0,24	357	6,96	0,22	400	7,17	0,22	315	7,91	0,25
48	348	7,49	0,16	365	6,80	0,14	291	9,86	0,21	305	8,17	0,17
64	335	7,78	0,12	341	7,28	0,11	267	10,74	0,17	430	5,80	0,09
80	301	8,65	0,11	318	7,81	0,10	296	9,69	0,12	304	8,20	0,10

Table 3. Divide et Impera 200M

Cores	Packed			Packed Excl			Scatter			Scatter Excl		
	Time	Speedup	Efficiency	Time	Speedup	Efficiency	Time	Speedup	Efficiency	Time	Speedup	Efficiency
1	4262	1,00	1,00	4059	1,00	1,00	4115	1,00	1,00	4119	1,00	1,00
2	3963	1,08	0,54	3454	1,18	0,59	3808	1,08	0,54	3732	1,10	0,55
4	3180	1,34	0,34	2594	1,56	0,39	2777	1,48	0,37	2871	1,43	0,36
8	2010	2,12	0,27	1679	2,42	0,30	2201	1,87	0,23	2069	1,99	0,25
16	1366	3,12	0,20	1262	3,22	0,20	1253	3,28	0,21	1934	2,13	0,13
24	842	5,06	0,21	1015	4,00	0,17	1067	3,86	0,16	874	4,71	0,20
32	962	4,43	0,14	970	4,18	0,13	956	4,30	0,13	1097	3,75	0,12
48	764	5,58	0,12	889	4,57	0,10	801	5,14	0,11	777	5,30	0,11
64	724	5,89	0,09	739	5,49	0,09	658	6,25	0,10	647	6,37	0,10
80	806	5,29	0,07	655	6,20	0,08	693	5,94	0,07	726	5,67	0,07

Table 4. Divide et Impera 250M

Cores	Packed			Packed Excl			Scatter			Scatter Excl		
	Time	Speedup	Efficiency	Time	Speedup	Efficiency	Time	Speedup	Efficiency	Time	Speedup	Efficiency
1	8723	1,00	1,00	5557	1,00	1,00	6406	1,00	1,00	6100	1,00	1,00
2	7904	1,10	0,55	5746	0,97	0,48	8234	0,78	0,39	5747	1,06	0,53
4	3503	2,49	0,62	3768	1,47	0,37	3802	1,68	0,42	3016	2,02	0,51
8	2256	3,87	0,48	2258	2,46	0,31	2272	2,82	0,35	2273	2,68	0,34
16	1343	6,50	0,41	1640	3,39	0,21	1639	3,91	0,24	1441	4,23	0,26
24	1173	7,44	0,31	1434	3,88	0,16	1367	4,69	0,20	1590	3,84	0,16
32	1115	7,82	0,24	1149	4,84	0,15	1099	5,83	0,18	1443	4,23	0,13
48	1081	8,07	0,17	1020	5,45	0,11	936	6,84	0,14	874	6,98	0,15
64	936	9,32	0,15	932	5,96	0,09	823	7,78	0,12	817	7,47	0,12
80	956	9,12	0,11	871	6,38	0,08	1169	5,48	0,07	836	7,30	0,09

Table 5. Bruteforce 500K

Cores	Packed			Packed Excl			Scatter			Scatter Excl		
	Time	Speedup	Efficiency	Time	Speedup	Efficiency	Time	Speedup	Efficiency	Time	Speedup	Efficiency
1	6802	1,00	1,00	6802	1,00	1,00	6802	1,00	1,00	6802	1,00	1,00
2	7074	0,96	0,48	6285	1,08	0,54	5880	1,16	0,58	6230	1,09	0,55
4	3938	1,73	0,43	3502	1,94	0,49	3035	2,24	0,56	3499	1,94	0,49
8	1671	4,07	0,51	2705	2,51	0,31	2998	2,27	0,28	1662	4,09	0,51
16	952	7,14	0,45	1003	6,78	0,42	1974	3,45	0,22	867	7,85	0,49
24	723	9,41	0,39	734	9,27	0,39	616	11,04	0,46	678	10,03	0,42
32	510	13,34	0,42	778	8,74	0,27	467	14,57	0,46	454	14,98	0,47
48	414	16,43	0,34	405	16,80	0,35	331	20,55	0,43	309	22,01	0,46
64	313	21,73	0,34	311	21,87	0,34	277	24,56	0,38	231	29,45	0,46
80	242	28,11	0,35	236	28,82	0,36	242	28,11	0,35	221	30,78	0,38

Table 6. Bruteforce 1M

Cores	Packed			Packed Excl			Scatter			Scatter Excl		
	Time	Speedup	Efficiency	Time	Speedup	Efficiency	Time	Speedup	Efficiency	Time	Speedup	Efficiency
1	25771	1,00	1,00	25771	1,00	1,00	25771	1,00	1,00	25771	1,00	1,00
2	35780	0,72	0,36	25669	1,00	0,50	25658	1,00	0,50	25264	1,02	0,51
4	20698	1,25	0,31	14022	1,84	0,46	12789	2,02	0,50	12185	2,11	0,53
8	8994	2,87	0,36	7491	3,44	0,43	8077	3,19	0,40	6622	3,89	0,49
16	4040	6,38	0,40	4543	5,67	0,35	4425	5,82	0,36	3755	6,86	0,43
24	3156	8,17	0,34	2812	9,16	0,38	2926	8,81	0,37	2497	10,32	0,43
32	2028	12,71	0,40	2119	12,16	0,38	2151	11,98	0,37	2115	12,18	0,38
48	1599	16,12	0,34	1518	16,98	0,35	1528	16,87	0,35	1454	17,72	0,37
64	1235	20,87	0,33	1171	22,01	0,34	1117	23,07	0,36	1004	25,67	0,40
80	1022	25,22	0,32	964	26,73	0,33	935	27,56	0,34	846	30,46	0,38