University of Camerino

School of Science and Technologies

Master in Computer Science - Parallel and Distributed Programming

KNN implementation using MPI standard

Group members

Diego Belardinelli

Email: diego.belardinelli@studenti.unicam.it

Franco Suelgaray

Email: franco.suelgaray@studenti.unicam.it

UNICAM Supervisors

Prof. Andrea Polini

A.A. 2023/2024

1 Introduction

K-Nearest Neighbours, also known as simply K-NN, is a method used mostly in Machine Learning systems where non supervised approaches are needed to classify sets of values. Given a previously generated set of points that are (uniformly) distributed over an n-dimensional space, a specified number of neighbours K, and a distance metric, the method computes for each one of the said points the K nearest points in their neighbourhood.

The aim of the present project is to design and develop a parallel version of the K-NN algorithm using the MPI (Message Passing Interface) standard. Moreover, a sequential version and three parallel ones will be presented so as to compare and prove the advantages of a parallelized approach in algorithms of this nature. Throughout the different sections of the current report, a detailed description of the design and implementation decisions will be progressively introduced, beginning from the sequential implementation that will later serve as a base case for the following three parallelized versions. As a means of justifying the decisions made in each step of the development, for each of the algorithms described, tables with time tracking results will be provided and attached in the Appendix sections.

2 Sequential Algorithm

To provide a clear view of the advantages that parallelized implementations of algorithms bring (when the computational resources are available), this section will focus on the description of the sequential implementation of the K-NN algorithm, that will later be compared to the parallel versions in the following three sections. We will be first introduce the main design decisions taken before writing the code itself. Secondly, we will provide a brief description of the functions implemented for the computation of distances, and finally we will present summarized results and conclusions on the observed values and patterns found in the data.

2.1 Introduction

Given a set of generated points in a 3-dimensional space, the objective of the algorithm is to keep a list for each of them with the K nearest other points. As a distance metric we will use the *Euclidean distance*:

Given the points in the space $v_1 = (x_1, y_1, z_1)$ *and* $v_2 = (x_2, y_2, z_2)$*, the Euclidean distance can be computed using the following equation:*

$$d = ||v_1 - v_2|| = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

This algorithms presents a challenge when it comes to repeated data computation due to the way points have to be compared against each other so as to build the neighbours lists. This has also a direct impact on the distribution of data and the way it should be stored in memory. Throughout this section we will show how this challenge drove us to take design decisions that prioritize performance over code simplicity.

The present sequential version of the algorithm, written in the file `sequential-knn.c`, takes four external parameter provided by the user:

```
./sequential-knn N K BS L

N:  Number of points to generate
K:  Number of neighbours to account for
BS: Size of block
L:  Upper boundary for point generation
```

These parameters will be progressively introduced in the following sections where further details will be provided.

2.2 Memory Allocation and Data splitting

As it is already clear for every algorithm, storage and computation complexity should always be kept as low as possible. This subsection presents the main data structures the algorithm works with to accomplish its task and how their definitions and allocation can impact on the performance. Our implementation of the K-NN sequential algorithm consists of four main data structures:

1. ***Points array:***
   Generally, a set of points could be thought as a $n \times m$ matrix where each row represents a point of m dimensions and whose coordinates are determined by each respective column. Since C compiler saves matrices in memory by arranging the rows in a consecutive manner, we decided to imagine the points matrix as a simple array which can be accessed through a single pointer that we defined as `double *points`. Afterwards, given that each point has three coordinates, `3xN` memory slots of `double` values where allocated for their later storage. This way of storing the points matrix helps maintaining spacial locality while executing the computations.

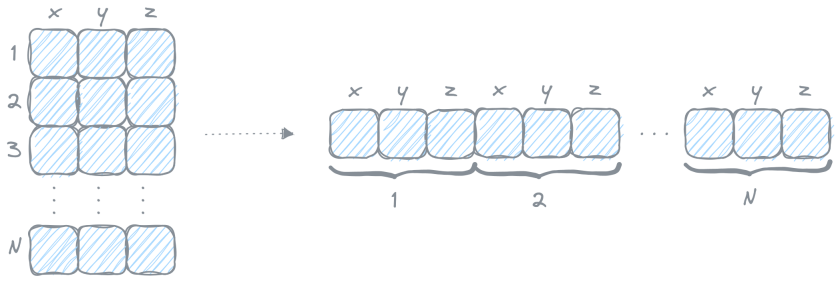

Figure 1: Points data structure from matrix to array disposition

2. ***Neighbours matrix:***
   The fact that all points have to define a neighbours list leads us to think in a data structure with a matrix format, where the rows correspond to each point and the columns to the neighbours of the corresponding point. Then, to prevent cache misses and maintain spacial locality, we rearranged this matrix as an array, similarly to the previous data structure, with each row next to the other in memory.

   To store the `K` nearest neighbours we use a pointer `int *near_neighs` which stores the memory address to a 1-dimensional array with `NxK` allocated slots of memory. Every consecutive set of K slots stores the ids of the points which are neighbours for another specified point. The way of enumerating the points with ids was directly taken from the order in which they

were stored in the `points` structure. Figure 2 shows how the matrix format gets converted into the array disposition in memory.

3. ***Minimum distances matrix:***
   This data structure stores the updated minimum distances throughout all the computations of the algorithm. It helps identify if a point P1 should be added to the neighbours list of another point P2 given the distance between them. In the same manner we thought the `near_neighs` data structure, we defined a pointer `double *min_dist` that also allocates `NxK` memory slots where the distances between each point and its neighbours is going to be stored. The following Figure 2 also shows a graphical description of the arrangement of the data structure in memory.

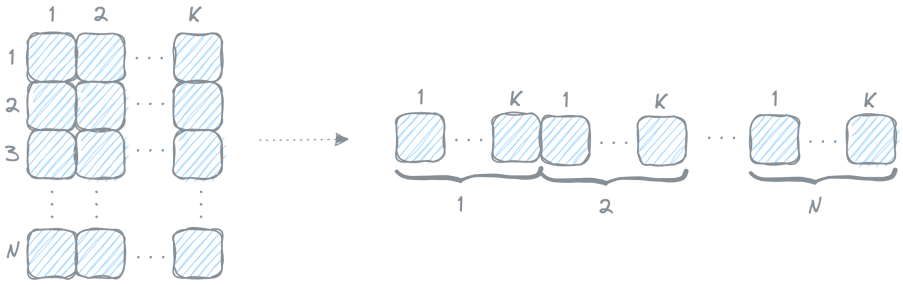

Figure 2: Neighbours and minimum distances data structures from matrix format to array disposition

4. ***Distances matrix:***
   Lastly, to temporarily store the results of the computations, we defined a data structure `double *distances`. Given that the distance from each point has to be calculated to every other one, we initially thought of a `NxN` matrix where each cell ij represents the distance between the points i and j. In this moment it is important to highlight that this matrix is clearly symmetric and where all values in the diagonal are zeros.

   Even though the code can gain some complexity when considering a symmetric matrix, taking advantage of this property usually brings performance and storage improvements. Avoiding repeated computations for each ij and ji pairs of cells improves performance, and ignoring repeated values when storing them, reduces memory usage.

   Moreover, to guaranty the highest spacial locality as possible, we considered partitioning the computations into smaller batches, where we iteratively calculated the distances between *subsets* of points. The size of these subsets or blocks is given by the parameter `BS` that is provided by the user. This is the reason why we reduced the original `NxN` matrix into a smaller

one with `BSxBS` that is reused when each point of a block gets compared to the ones in another block.

This approach helps reduce considerably the amount of memory allocated for the distances data structure, it helps maintain spacial locality and prevents replication of data and unnecessary computations. The following diagram (Figure 3) shows a graphical illustration of the distances matrix in two different scenarios:

- *(left)* When a block of points has to be compared against itself, the data structure is not used in its entirety due to the fact that, in this case, computing the distance of a point P to itself is not needed, and neither are the symmetric portions of the matrix.
- *(right)* When a block A of points has to be compared against another block B the entire structure is used.

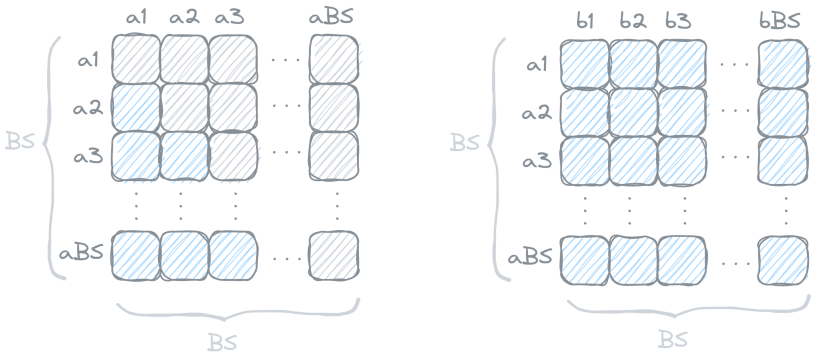

Figure 3: Distances data structure in both possible usage scenarios

Finally, this matrix is also accessed by a pointer and the rows are arranged consecutively in memory, which helps spacial locality.

2.3 Data generation

The function `generate_points` is the responsible to take the number of points `N` the user wants to generate, the data structure to save them in and the limit `L` that serves as an upper boundary value for the points generation. This particular function provides a uniformly distributed random way of populating the points data structure with positive integer values that vary from `0` to `L`. The following is the method's definition:

```
void generate_points(int numPoints, double *points, int limit)
```

```
numPoints: number of points N
points: pointer to array of all point coordinates
limit: upper boundary value L
```

For each of the `N` points, the function generates and saves three coordinates that represent x, y and z coordinates respectively.

2.4 Computation

The execution of the algorithm begins with the random generation of uniformly distributed points that are stored in the `points` array. Afterwards, given the block size `BS` provided by the user, the amount of blocks in which the points array has to be split into is computed by dividing the total number of points `N` by the block size `BS`. Then, two main operations have to be computed to accomplish the desired results:

- ***Distance computations:***

  Given two blocks of points and the matrix of distances, for each point P in one of the blocks, the distance from P to every other in the second block is computed using the Euclidean distance as metric and is stored inside the `distances` matrix in the corresponding cell of P's row. The function that solves this operation is:

  ```
  void computeDistances(double *points_a, double *points_b,
                        double *distances, int num_points,
                        bool against_self)
  ```

  ```
  points_a: pointer to initial position of block A
  points_b: pointer to initial position of block B
  distances: pointer to initial position of distance matrix
  num_points: number of points to be compared
  against_self: boolean that indicates if block A = block B
  ```

  The only exception where the execution of this function varies is when the blocks are the same and the flag `against_self` is `true`. In that case not all the distances are calculated.

- ***Ordering of neighbours:***

  After the distances between two blocks are computed, we need to identify if the neighbours for each of the points need to be updated. For this operation we provide two different functions: `updateNeighboursRowAx` and `updateNeighboursColAx`. The following is the definition for the first of functions, which is identical to the second one:

```
void updateNeighboursRowAx(int *neighbours, double *min_distances,
                           int num_neigh, double *distances,
                           int num_points, int points_b_first_idx,
                           bool against_self)
```

```
neighbours: pointer to the neighbours matrix (possibly with an offset)
min_distances: pointer to the minimum distances matrix (possibly with an offset)
distances: pointer to initial position of distance matrix
num_points: number of points to be compared
points_b_first_idx: first point index of block B
against_self: indicates if distances matrix has been computed with againts_self = true
```

  The first function works on a Row-axis manner, going over each row of the distances matrix and identifying if the distance in each cell is less than those stored for the neighbours of the particular point. To verify if the list of neighbours has to be updated, the algorithm performs a binary search through the array of minimum distances to determine the most probable position where the new distance could be stored. Once this position is found the algorithm verifies if the stored values from that position onwards are grater or less than the new one and inserts it in the correct position.

  Instead of executing a linear search which has linear complexity $O(n)$, the algorithm uses binary search which improves the performance reducing the complexity to a logarithmic order of magnitude $log_2(n)$.

  Afterwards, the second function is executed to update the neighbours of the points that where set in the column positions when computing the distances. In that way, with only one data structure for the distances, we are able to update the neighbours for the points belonging to two different blocks in just two function calls (this prevents a large amount of context switches when calling functions).

Taking all the previously mentioned into account, the algorithm can be thought broadly in two main stages:

1. ***Self comparison:***

   For each block B, the algorithm performs the distance computations with the block B against itself and afterwards updates the neighbours going through the distances matrix in both directions (row and column wise). Figure 4 shows a visual representation of the comparisons between the different block in this first stage.

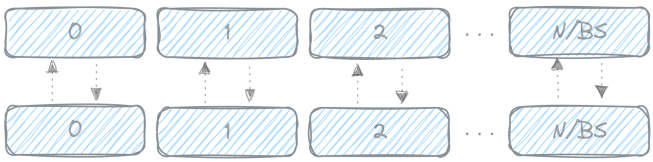

Figure 4: Diagram of blocks comparison in 1st stage of computation

2. ***Comparisons between different blocks:***

   After the self comparisons the algorithm goes through `N/BS - 1` steps where different block pairs are compared against each other, avoiding replicated computations. Then, analogously to the previous stage, each pair of blocks update their respective neighbours. The structure of these comparisons can be appreciated in the following Figure 5

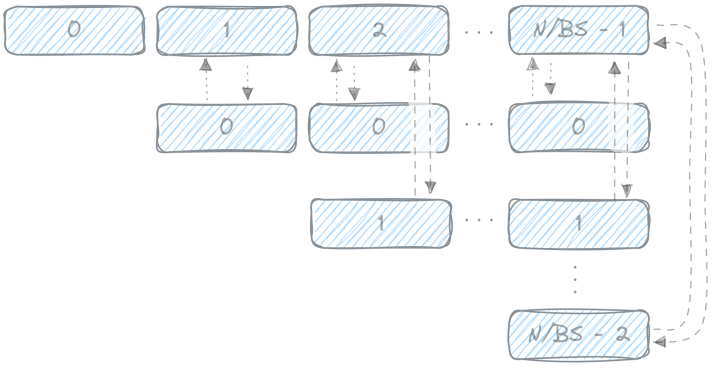

Figure 5: Diagram of blocks comparison in 2nd stage of computation

By the end of this stage, all distances are computed and all neighbours are correctly updated.

2.5 Time tracking

The sequential version of the algorithm not only serves us as the base for the following parallelized ones but it also gives us the possibility to compare the performance between the sequential and the parallelized versions and understand the advantages that a parallelized approach can bring to a computation-intensive algorithm.

To accomplish a precise comparison, we repeatedly tracked the time the algorithm took to complete its execution with different initial parameters, varying the dimension `N` of the problem (that is, the total number of points), the number of neighbours `K` to account for and the block size `BS` to partition the points dataset. To track the time we only considered where the computations happened, ignoring the time it takes for the algorithm to generate the data and allocate memory.

Finally, we built a script called `run-multiple.sh` that takes one parameter:

```
./run-multiple.sh K

K: number of neighbours to account for
```

It iterates over an array of dimensions (from `N=1024` to `N=65536`) and for each of them it executes an inner loop which tracks the time for different values of block sizes (starting with `BS=64`). For all the possible combinations the script executes the algorithm and returns the time tracking results.

We ran the script four times with different values of `K`: `K=5`, `K=10`, `K=15` and `K=20`

2.6 Results and Conclusions

All results obtained from the repeated executions of the algorithm can be seen in Tables A1, A2, A3 and A4. The values are expressed in seconds and the smallest durations are highlighted in bold.

Taking these values into account and analysing the pattern they show, we can clearly identify that the chosen block size can have a critical impact on the overall performance of the algorithm. So much so that it can almost triple the duration as in the case with `K=20`, `BS=65536` and `N=65536`. One obvious approach to the sequential algorithm would be to process all data as a single block, but it can be seen in the results that this is actually the configuration where the algorithm performs its worst in almost all executions.

With spacial locality playing an important role in this algorithm, we can highlight the relevance of the specific hardware architecture that runs underneath the software layer. In the case, it is straightforward to determine that the block size value which fits best Sibilla's cache in the majority of the cases is `BS=128`.

3 Parallel Algorithm V1

Taking the sequential algorithm and its characteristics described in the previous section (Section 2), we will introduce a first approach to a parallelized version using MPI standard for message passing. Data structures, computation flow and results are going to be presented throughout this section, aiming to identify if the parallelization of the algorithm can achieve better performance.

3.1 Introduction

As in the previous implementation shown in Section 2, the distance metric used is the *Euclidean distance* for 3-dimensional points in space.

One aspect to highlight in this version of the parallelization is that block sizes (`BS`) are not used to split the points data. The current implementation of the algorithm, written in the file `parallel-knn-v1.c`, takes three external parameter provided by the user:

```
./parallel-knn-v1 N K L

N:  Number of points to generate
K:  Number of neighbours to account for
L:  Upper boundary for point generation
```

On the other hand, data generation procedure is unchanged, keeping the same behaviour as described in Section 2. The only detail to highlight regarding this functionality is that is handled only by the process with rank 0, which from now on we will name `COORDINATOR`.

3.2 Memory Allocation and Data splitting

Data structure formats are mostly conserved in this parallel version of the algorithm, but with the highly important difference that, in this case, the memory allocation is done in every single one of the defined processors. All of them share five main data structures as shown in Figure 6.

- ***Points array:***
  In the same manner the array of generated points was thought in the previous section, in this parallel version, the data structure is stored as an array of memory slots which can be accessed through the pointer `double *my_points`.

  In this case, the `COORDINATOR` process allocates all `3xN` slots of memory, whereas the rest of processes reserve only `3 x stripsize`. This new value `stripsize` is the result of dividing N by the total number of defined processes, which will be used by all processes to compute the portion of data they have to consume and produce. The `COORDINATOR` is the responsible to

generate all the points and store them in the array. After the allocation of memory in the other processes, the `COORDINATOR` distributes an equal number (`stripsize`) of points to each of the others so as to split the overall computation.

- ***Neighbours and Minimum distances matrix:***
  Taking into account that each process has its own set of points to account for, it is reasonable for each of them to have matrices for neighbours and minimum distances where the process can keep track of the respective neighbours of its own points and the distances to them.

  Both data structures, `near_neighs` and `min_distances`, are defined in the same way they were specified for the sequential algorithm in Section 2. They are stored as continuous arrays of values to prevent as many cache misses as possible and to favour spacial locality.

  As shown in Figure 6, these data structures have the same dimensions between them, but they vary btween the `COORDINATOR` and the rest of processes. While process 0 allocates `NxK` slots of memory, the other processes allocate just a portion of `stripsize x K`. During the main computation of distances and ordering of neighbours these structures are used as storage for intermediate results. Afterwards, when the final results are obtained, all data is gathered in the structures defined by the `COORDINATOR` which holds the complete set of final results.

- ***Distances matrix:***
  As temporarily storage for the results of the computations, we defined a data structure `double *distances` in each of the processes, and whose dimensions are identical across all of them (including the `COORDINATOR`). This data structure is thought in the the same way as in the sequential algorithm, but given that block sizes are not considered in this opportunity, the dimensions are `stripsize x stripsize`.

  As in Figure 3 show, and as it was previously explained in section 2, this data structure can be used in two different scenarios:

  - When a block of points has to be compared against itself.
  - When a block A of points has to be compared against another block B.

- ***Other points array:***
  As it will be introduced in the following section, for each stage of the algorithm, different portions of the set of generated points are sent to each process to compute a subset of the overall results. Since every process has its own set of points stored in the `my_points` data structure, another set of memory slots is allocated to temporarily store these new incoming points. This allows the processes to store both set of points for the computation of distances.

  The dimensions of this new array are `3 x stripsize` and it can be accessed by the pointer `double *other_points`. As shown in the Figure 6, both `my_points` and `other_points` have

the same dimensions in all processes except for the `COORDINATOR`, where the former is bigger, storing all generated points.

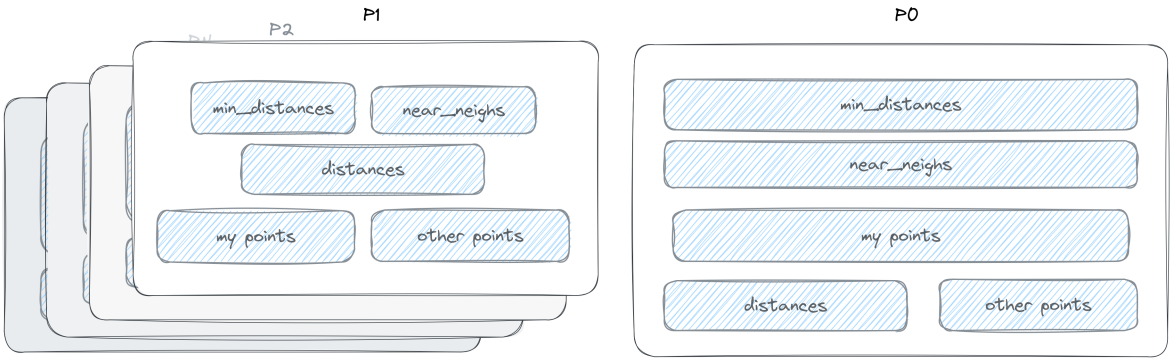

Figure 6: Data structures allocated in each processor

3.3 Computation

The computation of this parallel algorithm can be separated into three main stages:

1. ***Data Scattering:***
   The first thing after the allocation of memory and the generation of points, the `COORDINATOR` distributes the points by equally assigning a portion of size `N/P` to each of the processes. This is accomplished in a performant way by utilizing the optimized `MPI_Scatter` function.

   All these points are stored in the `my_points` data structure in every process.

2. ***Computation of Distances and Sorting of Neighbours:***
   Once the points were stored in the processes, for `P` steps, the `COORDINATOR` saves a portion of the generated points into the `other_points` data structure which is then broadcasted to all processes by using the function `MPI_Bcast`.

   After receiving the other set of points, each process computes the distances form their own points to the new ones by using a similar function to the one was introduced in Section 2. Then, the new neighbours are identified and they are sorted using both binary search and a linear sorting. For this last stage we use the function `updateNeighbours` and store the results in `min_distances` and `near_neighs` .

3. ***Data Gathering:***
   Lastly, when all processes have finished their computations the results are gathered in the `COORDINATOR`'s `min_distances` and `near_neighs` arrays by making use of the function `MPI_Gather`.

These steps can be graphically seen in Figure 7

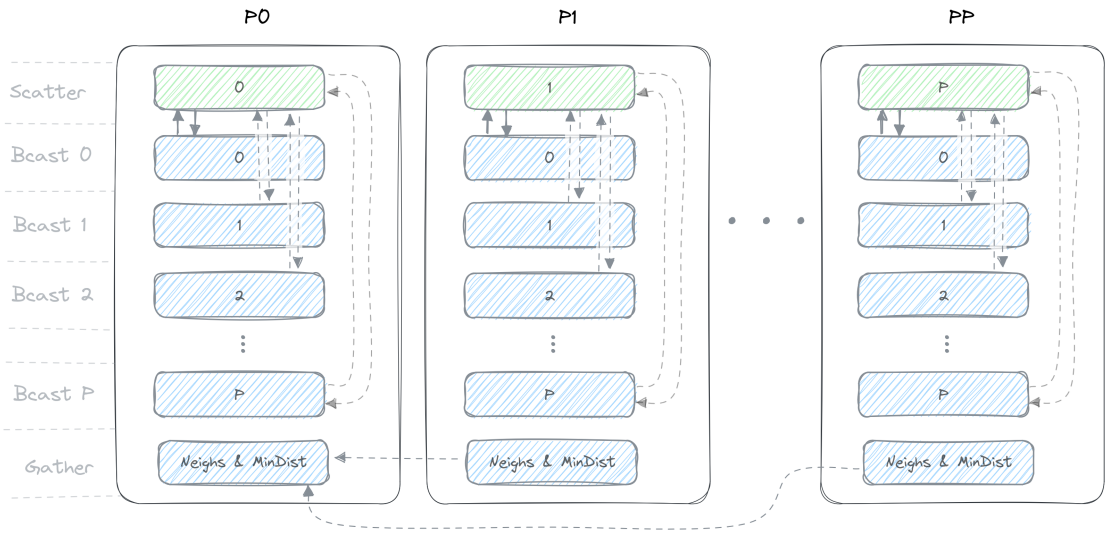

Figure 7: Computation diagram for 1st version of parallel algorithm

3.4 Time tracking

To track the time it takes the algorithm to complete its computations we used two functions:

- `MPI_Wtime` to mark the beginning and end of each process, by executing it once at the beginning of the computation and once more at the end. Then we subtracted them to determine the total duration.
- `MPI_Reduce` to identify the maximum duration amongst all processes. The result is saved in the `COORDINATOR` process.

We repeatedly executed the algorithm and took the time for different values of `N` and increasing amounts of processes. All results can be seen in Tables B5, B6, B7 and B8.

This same approach was taken to track the time for the following two other versions of the algorithm.

3.5 Performance Analysis

During this section we will present a set of results on two different metrics that measure the parallel algorithm's performance with respect to the sequential one, *Speedup* and *Efficiency*.

Taking into account the duration for the sequential algorithm when `BS=128` (we took this same set of results for the following two other implementations) that can be seen in Appendix A, for all different values of `K` and all dimensions, we computed both measurements by following the equations

$$Speedup(n,p) = \frac{T_{seq}(n,p)}{T_{par}(n,p)}$$

$$Efficiency(n,p) = \frac{Speedup(n,p)}{p}$$

3.5.1 Speedup

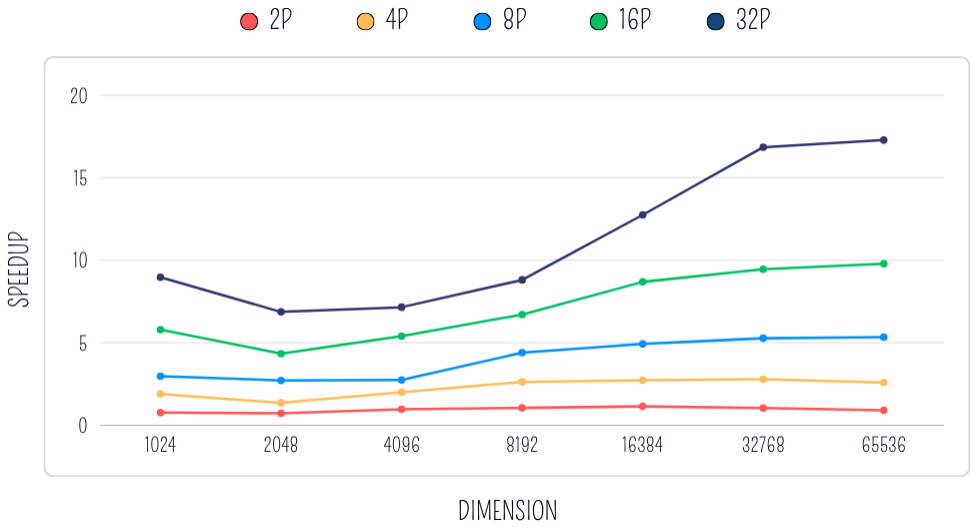

Figure 8: Speedup plot for 1st version of Parallel Algorithm - K=5

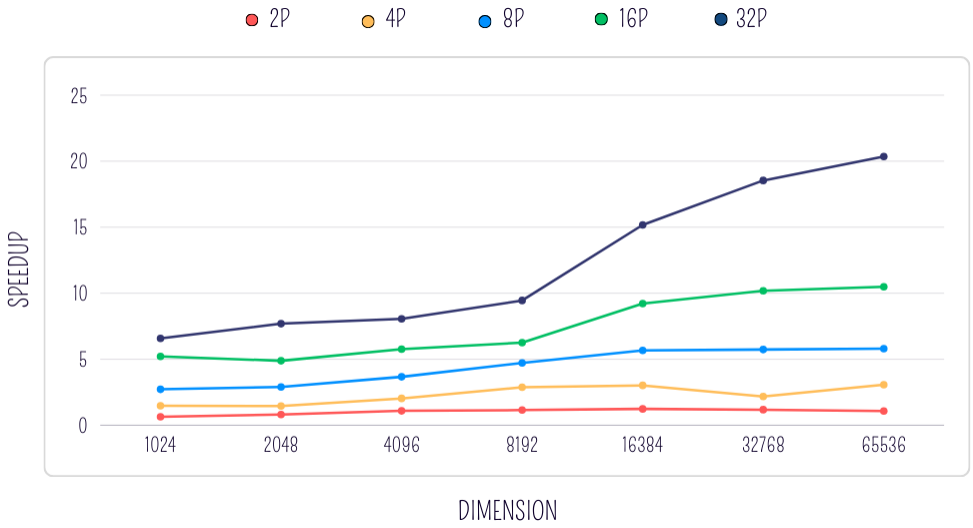

Figure 9: Speedup plot for 1st version of Parallel Algorithm - K=20

3.5.2 Efficiency

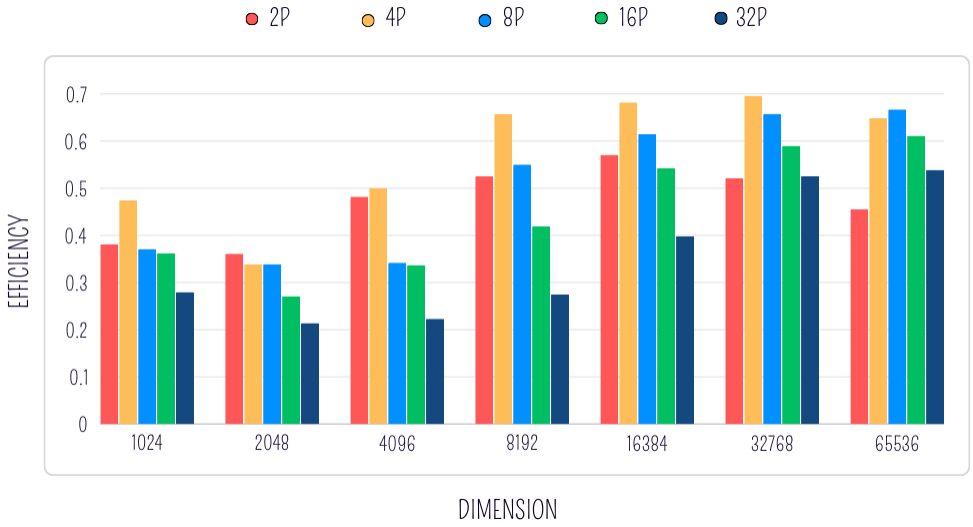

Figure 10: Efficiency plot for 1st version of Parallel Algorithm - K=5

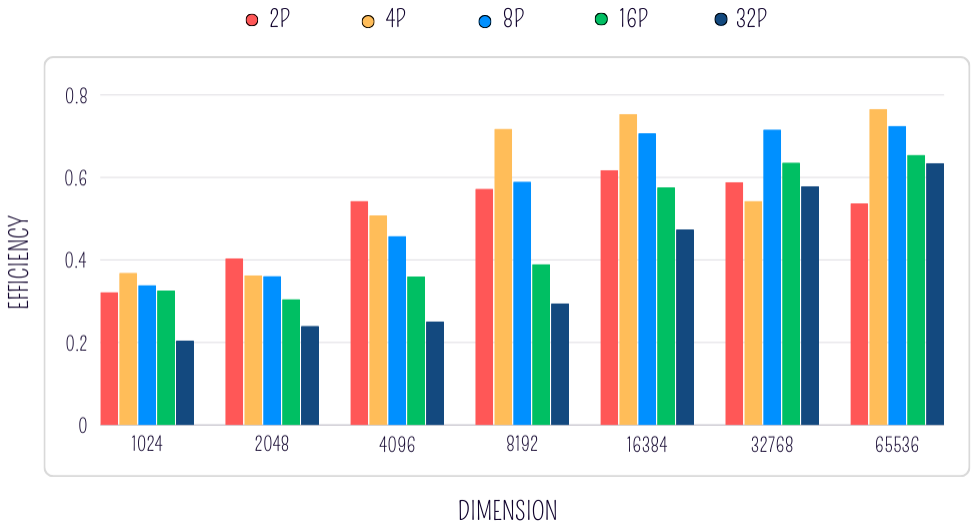

Figure 11: Efficiency plot for 1st version of Parallel Algorithm - K=20

3.6 Results and Conclusions

In Appendix A the reader can see all the results for the duration of the algorithm's execution.

For this implementation we can highlight the fact that, despite replicating information and computations, the algorithm achieves considerable improvements when the number of processors and the dimension of the problem increase. The results on the Speedup give us a clear view of how many times faster this algorithm is with respect to the sequential one, and we can identify that, given a number of processes `P`, and for a big amount of points, the time can be reduced a little bit more than `P/2` times.

On the other hand, when the number of processes is low, the replication of data and the communications that have to be established have a great impact on the performance, getting worse results than the sequential implementation.

4 Parallel Algorithm V2

Given that the previous first parallel implementation of the K-NN algorithm suffers from the duplication of the distances computations, this present implementation tries to avoid this replication of data by introducing more collective communications.

We will see that despite the fact that the replication is avoided, the addition of more synchronizations points for processes communications has a great impact on the overall algorithm's performance.

4.1 Memory Allocation and Data splitting

The data is split in the same manner as in the first parallel by assigning a block of size `N/P` to each process. This is accomplished by using `MPI_Scatter` with the `COORDIANATOR` process as the source of data.

Moreover, the differences that the 1st and 2nd implementations have in terms of data structures are:

- The second algorithm increases the dimension of the `distances` matrix to `N*stripsize` (where `stripsize=N/P`)
- The second algorithm allocates another matrix called `dist_mat_transp`. This new matrix, meant to store the transpose of the distance matrix, is accessed through a pointer and has dimensions `stripsize x stripsize` (where `stripsize=N/P`).

4.2 Computation

A particular aspect of this case of algorithm is that, despite being an SIMD program, not all processes execute the same computations at the same time.

After the generated points are equally distributed in blocks to all processes, the `COORDINATOR` broadcasts the its own block of points to all processes through the `MPI_COMM_WORLD` communicator so they compute the distances from their own points to those in this new set. Afterwards, all results are gathered in the `COORDINATOR`'s `distances` matrix, and, for doing that, the other processes have to transpose their result matrix so as to correctly fit the receiver's data structure.

Then, for `P-2` steps, the process `i` broadcasts its own block to the rest, excluding those whose `rank < i`, through another new communicator which contains only the last `P-i` processes. They compute the distances to the `i`-th block and when all results are obtained, the distances matrices are transposed and gathered in the `i`-th process `distances` matrix.

After each process receives the results for the distances to their own points, they sort the neighbours by going through the entire `distances` matrix. This is done while the rest of processes may be computing the distances to more blocks of points. By separating and gathering the data in this manner, we can perform the computations only in the half of the entire distances matrix, avoiding the duplication of data. All this procedure can be graphically interpreted with Figure 12

Finally, when each process finishes sorting the neighbours and minimum distances to their own points, all results are gathered in the `COORDINATOR`'s `near_neighs` and `min_distances` arrays, identically as in the previous implementation.

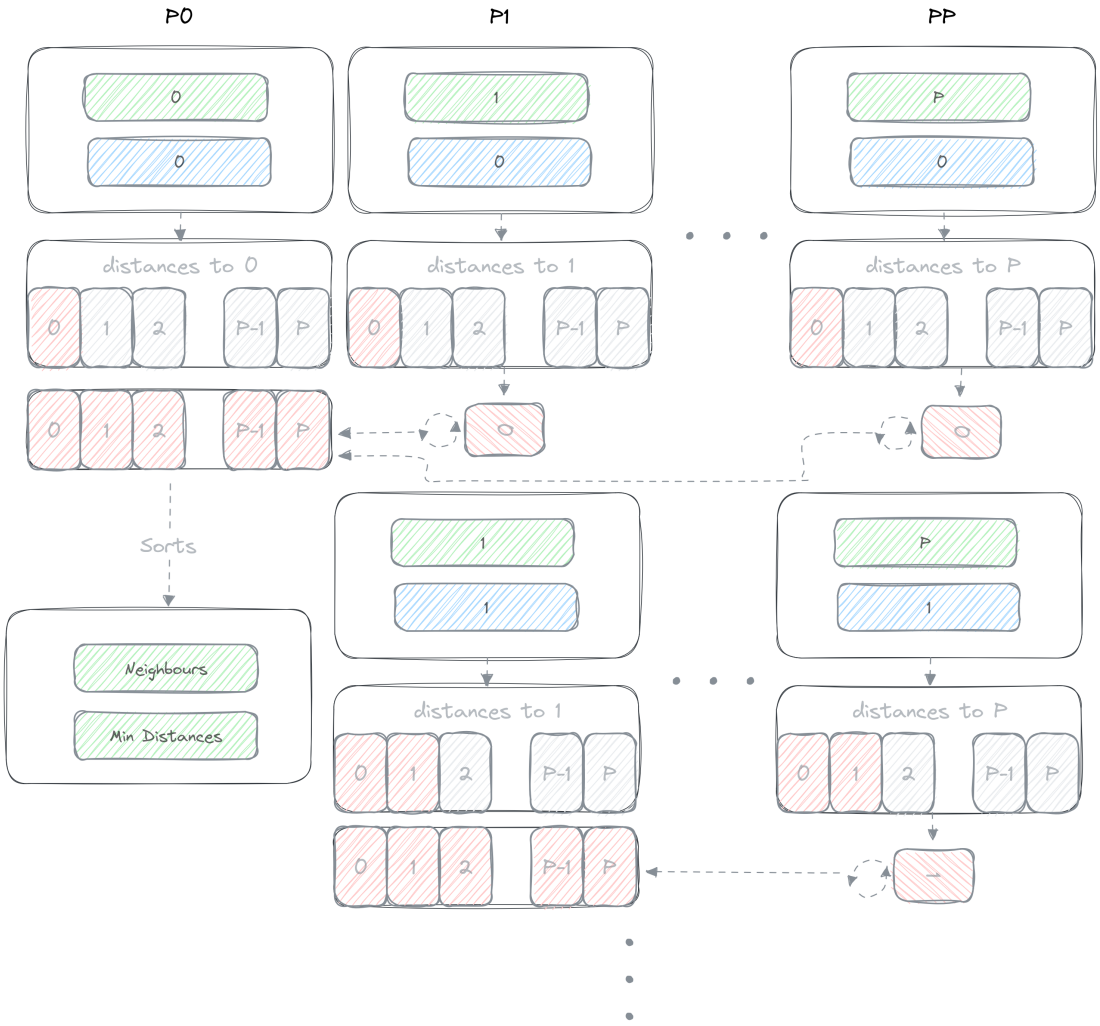

Figure 12: Computation diagram for 2nd version of parallel algorithm

4.3 Performance Analysis

Just like in the previous section, we will compute the Speedup and Efficiency of this implementation to analyse its performance against the sequential version. All results can be seen in the tables in Appendix C.

4.3.1 Speedup

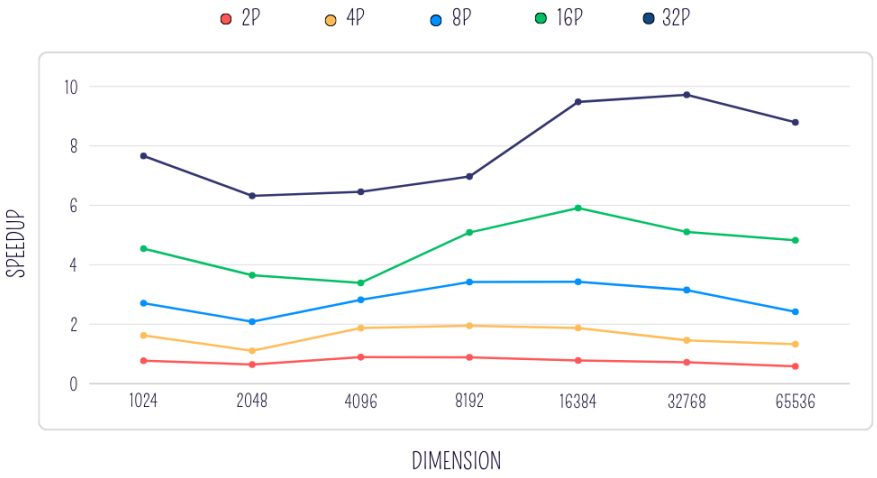

Figure 13: Speedup plot for 2nd version of Parallel Algorithm - K=25

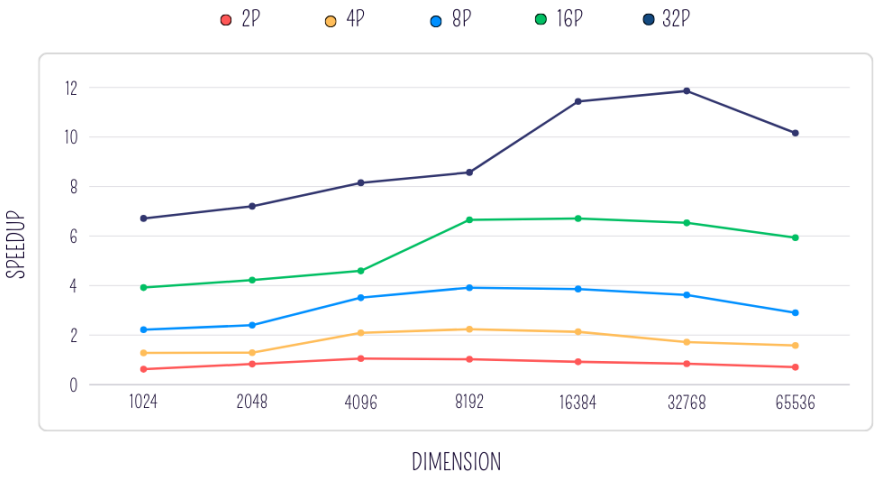

Figure 14: Speedup plot for 2nd version of Parallel Algorithm - K=20

4.3.2 Efficiency

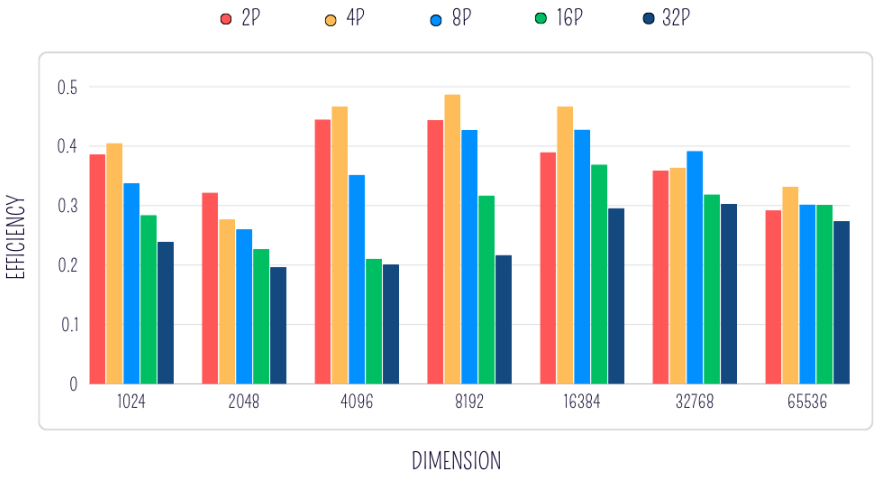

Figure 15: Efficiency plot for 2nd version of Parallel Algorithm - K=5

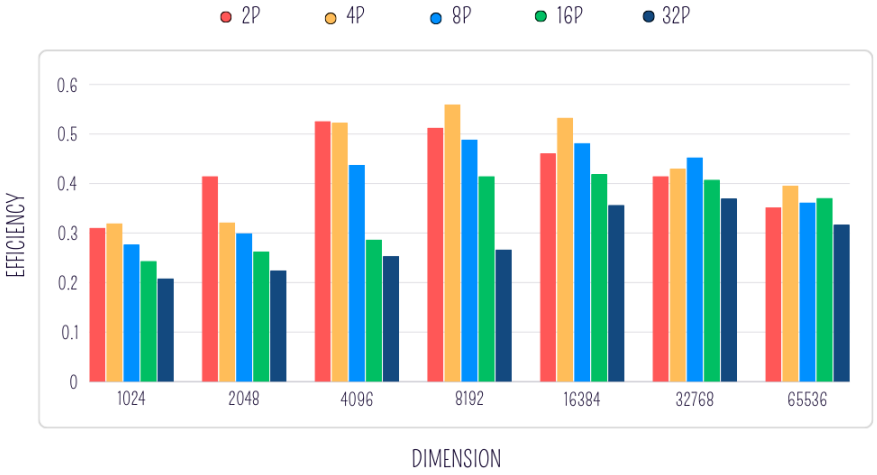

Figure 16: Efficiency plot for 2nd version of Parallel Algorithm - K=20

4.4 Results and Conclusions

As mentioned in the previous section, all results for the duration of execution can be seen in Appendix A, specifically Tables B9, B10, B11 and B12.

After analyzing these results and comparing them to the ones obtained for the first implementation, we can clearly see that they got worse by introducing more communications in this second version. This behaviour can be attributed to:

- The time to establish communication and transferring or receiving data to other processes, either by broadcasting or gathering.
- The need for synchronization to communicate the information can lead to processes with idle time after computing the distances.
- Processes can suffer of idle time when they finish sorting their neighbours and they have to wait for the rest so as to gather all results in the `COORDINATOR` process.

The decrease in performance can be clearly identified in the speedup plots, where the improvement in duration can not even reach half of the optimal timing. It is always $Speedup(n, p) < p/2$. This has a clear impact on the Efficiency and we can see that for `P=2`, the algorithm performs worse than the sequential.

Finally, this makes us reflect on the idea that, given a particular situation, sometimes it is better to replicate data in order to avoid unnecessary idle time and excessive communications.

5 Parallel Algorithm V3

Now that we have made two previous implementations of the parallel algorithm, this third one tries to improve them both by taking into account the aspects highlighted in the other sections.

As we have seen, the first parallel algorithm had a main weakness point which was the replication of distances computations. On the other hand the second algorithm avoided all replication of data, but at the expense of more communications between the processes and not all them where in the same computation stage at the same time. This increase in communications had a big impact on the performance of the second implementation.

Having this in mind, we implemented a third algorithm preserving the replication but reducing the number of communications between the processes.

5.1 Memory Allocation and Data splitting

Our third implementation takes the first parallel algorithm as a base, inheriting almost all data structures with the same dimensions. The only differences are:

- This version does not include the `other_points` data structure for all processes.
- This version increases the dimension of the `my_points` data structure in all processes to `Nx3`, so all generated points can be stored.

The following Figure 17 shows the distribution of the data structures throughout all processes.

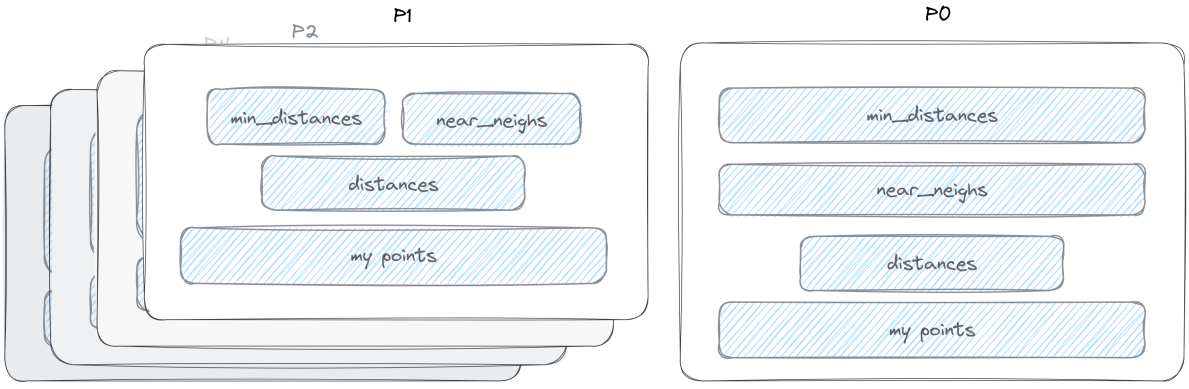

Figure 17: Data structures in all processes for third implementation

5.2 Computation

An easier computation was implemented in this case. Beginning with a broadcast of all points from the `COORDINATOR` process, all processes get the entirety of the generated dataset in only one step, storing it in the `my_points` array.

Afterwards, each process gets a partition/block of size `N/P`. The process `i` computes the distance of points in the `i`-th block to all the rest. The comparisons are made in batches of size `N/P`, and for each step the computed distances are used to sort the neighbours of the process' points.

Finally, when all processes have finished sorting all neighbours of their own points, the results are gathered in the `COORDINATOR` process and stored in the `near_neighs` and `my_distances` arrays.

All these steps can be graphically seen in Figure 18

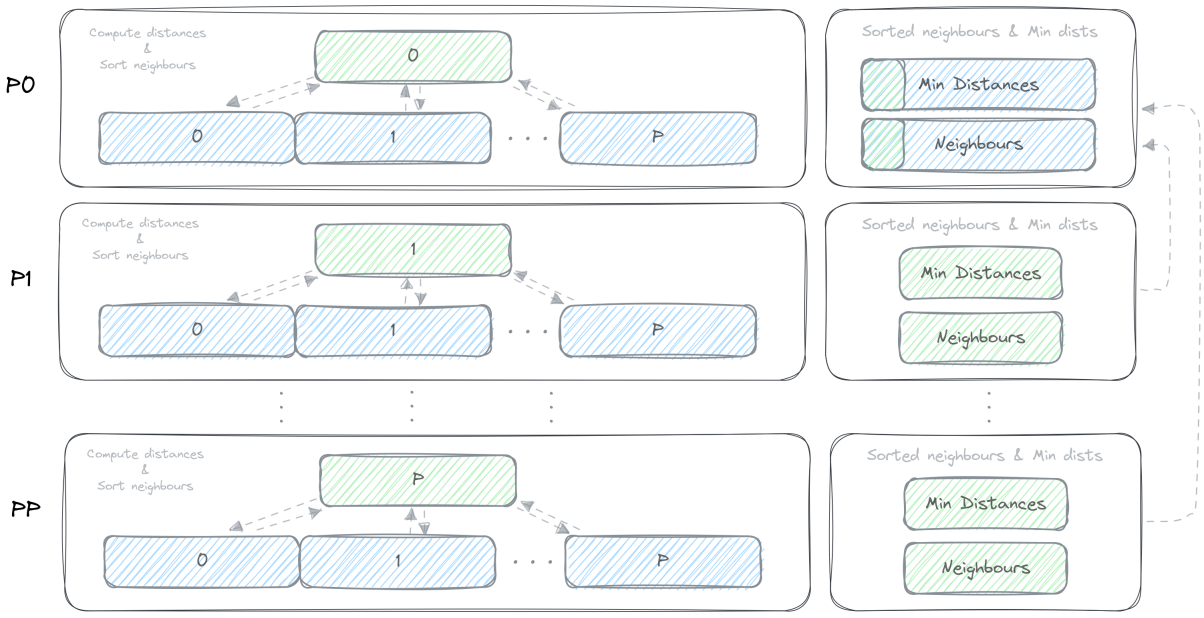

Figure 18: Computation diagram for 3rd version of parallel algorithm

5.3 Performance Analysis

In this section we present the Speedup and Efficiency results for the third parallel implementation in form of plots. The tables can be seen in the Appendix C.

5.3.1 Speedup

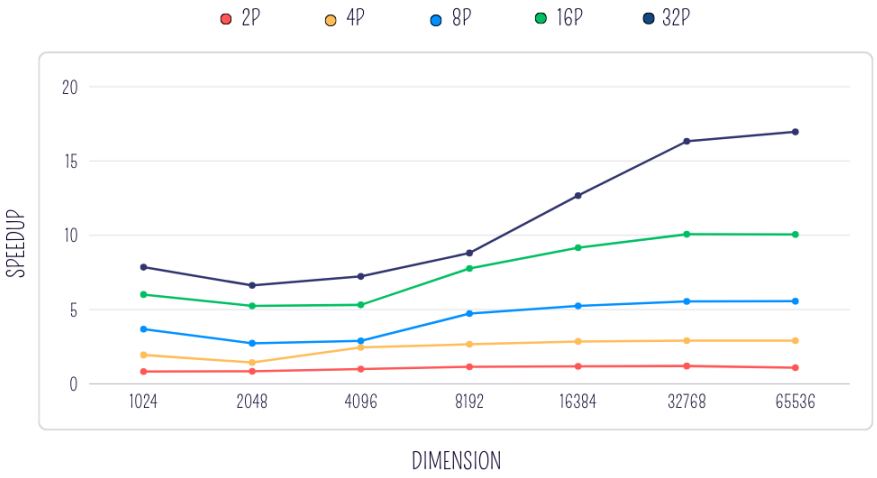

Figure 19: Speedup plot for 3rd version of Parallel Algorithm - K=25

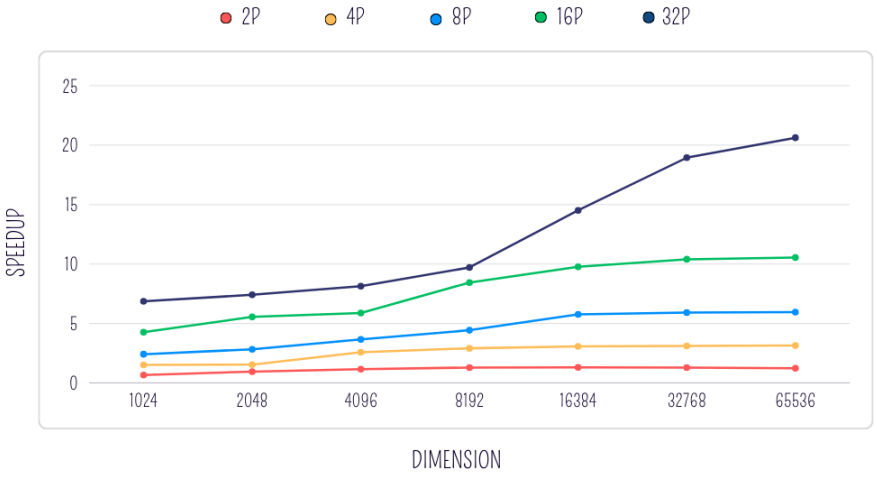

Figure 20: Speedup plot for 3rd version of Parallel Algorithm - K=20

5.3.2 Efficiency

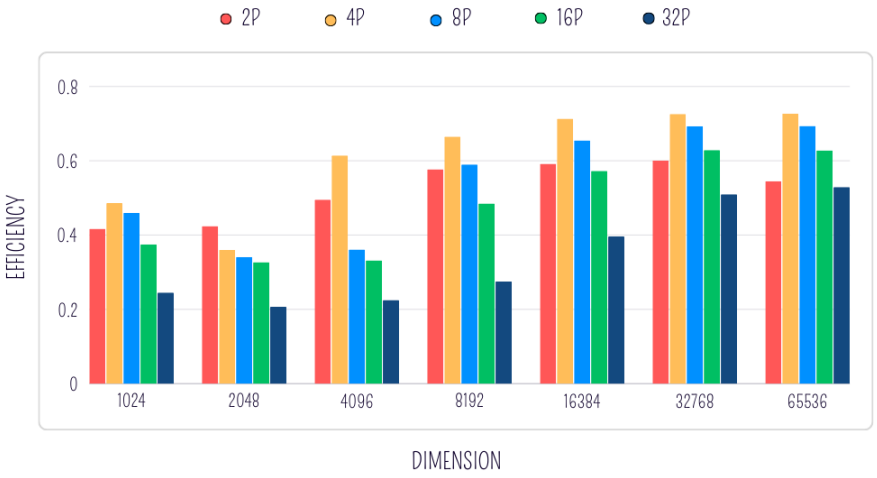

Figure 21: Efficiency plot for 3rd version of Parallel Algorithm - K=5

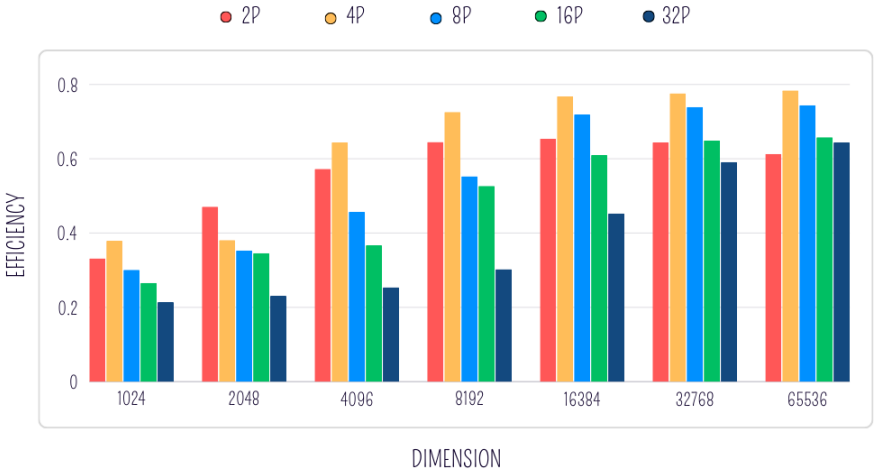

Figure 22: Efficiency plot for 3rd version of Parallel Algorithm - K=20

5.4 Results and Conclusions

Having a look at Tables B13, B14, B15 and B16 we can identify that the results of this implementation are better than both first and second versions. We can attribute this behaviour to the reduction of time needed to synchronize the processes to communicate data. Since the collective communications happen only at the beginning and end of the algorithm, no process gets idle during the computation of the neighbours.

We can finally highlight two main disadvantages of this approach:

- We still have replication of the distances computations.
- There exist a threshold to the dimension of the problem from which the algorithm starts to perform worse. The amount of information to broadcast in the beginning of the algorithm gets too large to communicate to all processes. This decrease in performance can be spotted in both speedup and efficiency results when the number of points surpasses `N=32768`.

Appendix A

Sequential Algorithm Time Results

Computed neighbours K = 5										
N/BS	64	128	256	512	1024	2048	4096	8192	16384	32768
1024	0.144	0.095	**0.068**	0.073	0.076	-	-	-	-	-
2048	0.266	**0.265**	0.268	0.287	0.294	0.379	-	-	-	-
4096	1.058	**1.057**	1.067	1.139	1.168	1.529	1.635	-	-	-
8192	4.221	**4.182**	4.254	4.573	4.649	6.114	6.378	6.549	-	-
16384	16.916	**16.835**	17.031	18.327	18.615	24.598	25.606	25.856	25.763	-
32768	67.674	**67.388**	68.041	73.579	74.572	98.383	102.056	103.057	101.519	114.459
65536	**268.500**	269.372	270.537	290.518	297.986	394.392	405.422	411.562	402.1283	427.199

Table A1: Time results in seconds for sequential algorithm and K=5

Computed neighbours K = 10										
N/BS	64	128	256	512	1024	2048	4096	8192	16384	32768
1024	0.178	0.086	**0.082**	0.087	0.089	-	-	-	-	-
2048	0.325	0.324	**0.323**	0.338	0.345	0.433	-	-	-	-
4096	1.277	**1.278**	1.279	1.342	1.360	1.710	1.797	-	-	-
8192	5.095	5.075	**5.058**	5.355	5.428	6.865	7.508	7.231	-	-
16384	20.300	**20.251**	20.264	21.319	21.602	27.388	28.035	28.377	28.649	-
32768	81.042	**80.870**	80.920	85.815	86.357	109.556	112.540	113.483	112.462	123.786
65536	325.189	**321.615**	323.972	338.223	345.652	436.837	447.279	453.136	448.048	486.968

Table A2: Time results in seconds for sequential algorithm and K=10

Computed neighbours K = 15										
N/BS	64	128	256	512	1024	2048	4096	8192	16384	32768
1024	0.157	**0.086**	0.086	0.090	0.092	-	-	-	-	-
2048	0.333	0.333	**0.331**	0.348	0.353	0.470	-	-	-	-
4096	1.299	1.296	**1.295**	1.364	1.378	1.747	1.807	-	-	-
8192	5.128	**5.097**	5.123	5.387	5.447	6.887	7.105	7.246	-	-
16384	20.360	**20.271**	20.401	21.126	21.726	27.538	28.225	28.597	28.587	-
32768	81.335	**81.085**	81.277	85.361	86.677	110.076	112.755	113.666	112.619	141.837
65536	325.615	324.540	**324.521**	337.542	346.690	438.973	450.855	453.784	446.961	476.914

Table A3: Time results in seconds for sequential algorithm and K=1

Computed neighbours K = 20										
N/BS	64	128	256	512	1024	2048	4096	8192	16384	32768
1024	0.198	**0.096**	0.096	0.100	0.103	-	-	-	-	-
2048	0.367	**0.366**	0.368	0.382	0.390	0.482	-	-	-	-
4096	1.425	**1.424**	1.424	1.492	1.509	1.861	1.954	-	-	-
8192	5.608	5.606	**5.601**	5.920	5.955	7.348	7.680	7.802	-	-
16384	22.271	**22.214**	22.226	23.336	23.568	29.449	30.425	30.655	30.757	-
32768	88.969	**87.771**	88.438	93.150	94.786	116.886	121.306	121.410	120.101	133.145
65536	354.126	352.623	**352.437**	368.203	375.737	470.520	480.658	485.558	479.005	516.930

Table A4: Time results in seconds for sequential algorithm and K=20

Appendix B

Parallel Algorithm V1 Time Results

Computed neighbours K = 5							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.1244	0.3661	1.0945	3.9771	14.7302	64.6050	295.3860
4	0.0501	0.1953	0.5277	1.5910	6.1681	24.1970	103.7654
8	0.0320	0.0977	0.3857	0.9501	3.4188	12.7854	50.4592
16	0.0164	0.0611	0.1959	0.6237	1.9383	7.1320	27.5219
32	0.0106	0.0386	0.1478	0.4750	1.3210	4.0004	15.5924

Table B5: Time results in seconds for first version of parallel algorithm and K=5

Computed neighbours K = 10							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.1492	0.4011	1.3204	4.6031	16.9200	70.7005	325.174
4	0.0585	0.2278	0.5672	1.8064	6.9101	27.3727	109.363
8	0.0324	0.1142	0.3782	1.0522	3.6415	14.4980	57.414
16	0.0209	0.0617	0.2254	0.7675	2.3686	8.0505	31.682
32	0.0120	0.0415	0.1704	0.5490	1.4430	4.3236	16.494

Table B6: Time results in seconds for first version of parallel algorithm and K=10

Computed neighbours K = 15							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.1504	0.4353	1.2965	4.5872	18.1167	71.606	326.383
4	0.0600	0.2322	0.5629	1.8037	7.4779	27.4238	109.702
8	0.0326	0.1241	0.4532	1.0422	3.8157	14.4768	57.175
16	0.0177	0.0602	0.2310	0.7482	2.1796	8.1550	30.839
32	0.0119	0.0445	0.1652	0.5176	1.4857	4.3212	16.220

Table B7: Time results in seconds for first version of parallel algorithm and K=15

Computed neighbours K = 20							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.1488	0.4252	1.3091	4.8875	17.9550	74.3969	328.0727
4	0.0656	0.2520	0.7002	1.9496	7.3573	40.3582	114.9440
8	0.0353	0.1264	0.3884	1.1870	3.9225	15.3050	60.7498
16	0.0184	0.0750	0.2469	0.8963	2.4091	8.6163	33.6459
32	0.0146	0.0476	0.1767	0.5932	1.4637	4.7344	17.3263

Table B8: Time results in seconds for first version of parallel algorithm and K=20

Parallel Algorithm V2 Time Results

Computed neighbours K = 5							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.1230	0.3974	1.1869	4.7072	21.5440	93.6220	460.7569
4	0.0585	0.2310	0.5655	2.1453	8.9983	46.1958	202.7958
8	0.0351	0.1228	0.3749	1.2232	4.9094	21.3870	111.3181
16	0.0209	0.0702	0.2407	0.8221	2.8492	13.1991	55.8441
32	0.0124	0.0405	0.1637	0.6001	1.7757	6.9329	30.6262

Table B9: Time results in seconds for second version of parallel algorithm and K=5

Computed neighbours K = 10							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.1393	0.4716	1.3258	5.2559	21.5440	100.4003	486.3600
4	0.0677	0.2622	0.6230	2.4261	9.9182	49.0671	216.0162
8	0.0422	0.1383	0.3982	1.2967	5.2014	23.2813	117.0526
16	0.0216	0.0807	0.2728	0.8903	3.0715	13.0704	60.0387
32	0.0153	0.0471	0.1826	0.5024	2.0058	7.4000	31.6679

Table B10: Time results in seconds for second version of parallel algorithm and K=10

Computed neighbours K = 15							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.1426	0.3850	1.3249	5.1638	23.4627	99.8894	488.1181
4	0.0680	0.2640	0.6916	2.4097	9.9947	49.2733	208.3810
8	0.0398	0.1382	0.4104	1.3703	5.2327	23.3404	115.5997
16	0.0220	0.0799	0.2857	0.8569	3.2069	13.8608	59.1487
32	0.0151	0.0458	0.1814	0.5832	1.9276	7.4013	32.6448

Table B11: Time results in seconds for second version of parallel algorithm and K=15

Computed neighbours K = 20							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.1541	0.4400	1.3526	5.4631	24.0868	104.0932	500.6436
4	0.0749	0.2833	0.6798	2.5021	10.4131	50.9932	222.5732
8	0.0433	0.1525	0.4057	1.4315	5.7517	24.2180	121.5004
16	0.0245	0.0867	0.3101	0.8423	3.3114	13.4358	59.3992
32	0.0143	0.0508	0.1747	0.6541	1.9423	7.4011	34.7025

Table B12: Time results in seconds for second version of parallel algorithm and K=20

Parallel Algorithm V3 Time Results

Computed neighbours K = 5							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.1137	0.3120	1.0661	3.6216	14.2102	56.0603	246.9693
4	0.0488	0.1840	0.4303	1.5709	5.8948	23.1759	92.6232
8	0.0258	0.0969	0.3651	0.8847	3.2105	12.1446	48.4521
16	0.0158	0.0505	0.1988	0.5383	1.8362	6.6927	26.7959
32	0.0121	0.0400	0.1462	0.4749	1.3283	4.1276	15.8849

Table B13: Time results in seconds for third version of parallel algorithm and K=5

Computed neighbours K = 10							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.1297	0.4118	1.1853	4.0559	16.0166	63.4280	276.0841
4	0.0557	0.2146	0.6613	1.8172	6.7024	26.6726	106.0233
8	0.0304	0.1127	0.3804	1.1114	3.7270	15.0810	54.8354
16	0.0165	0.0682	0.2345	0.7053	2.1469	7.8237	31.6534
32	0.0122	0.0441	0.1587	0.4999	1.4194	4.3796	16.0060

Table B14: Time results in seconds for third version of parallel algorithm and K=10

Computed neighbours K = 15							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.1336	0.4281	1.1772	4.1728	16.0920	63.7580	278.3457
4	0.0575	0.1982	0.5973	1.8318	6.8322	26.5890	105.6451
8	0.0425	0.1143	0.4048	1.0233	3.6484	13.8984	55.0383
16	0.0183	0.0616	0.2315	0.6847	2.0861	8.9610	35.1236
32	0.0123	0.0434	0.1625	0.5472	1.3925	4.4280	17.7469

Table B15: Time results in seconds for third version of parallel algorithm and K=15

Computed neighbours K = 20							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.1445	0.3881	1.2424	4.3436	16.9784	68.0573	287.4240
4	0.0633	0.2398	0.5524	1.9291	7.2291	28.2704	112.4168
8	0.0398	0.1295	0.3889	1.2662	3.8588	14.8302	59.1761
16	0.0225	0.0660	0.2423	0.6641	2.2733	8.4404	33.4448
32	0.0140	0.0494	0.1751	0.5774	1.5307	4.6332	17.0974

Table B16: Time results in seconds for third version of parallel algorithm and K=20

Appendix C

Speedup and Efficiency - Parallel Algorithm V1

Speedup - Parallel V1 - K = 5							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.763	0.723	0.965	1.051	1.142	1.043	0.911
4	1.896	1.356	2.003	2.628	2.729	2.784	2.595
8	2.968	2.712	2.740	4.401	4.924	5.270	5.338
16	5.792	4.337	5.395	6.705	8.685	9.448	9.787
32	8.962	6.865	7.151	8.804	12.744	16.847	17.276

Table C17: Speed-up values for first version of parallel algorithm, taking K=5

Efficiency - Parallel V1 - K = 5							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.381	0.361	0.482	0.525	0.571	0.521	0.455
4	0.474	0.339	0.500	0.657	0.682	0.696	0.6487
8	0.371	0.339	0.342	0.550	0.615	0.658	0.6672
16	0.362	0.271	0.337	0.419	0.542	0.590	0.611
32	0.280	0.214	0.223	0.275	0.398	0.526	0.539

Table C18: Efficiency values for first version of parallel algorithm, taking K=5

Speedup - Parallel V1 - K = 10							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.576	0.807	0.967	1.102	1.196	1.143	0.989
4	1.470	1.422	2.253	2.809	2.930	2.954	2.940
8	2.654	2.837	3.379	4.823	5.561	5.578	5.601
16	4.114	5.251	5.669	6.612	8.549	10.045	10.151
32	7.166	7.807	7.500	9.244	14.033	18.706	19.498

Table C19: Speed-up values for first version of parallel algorithm, taking K=10

Efficiency - Parallel V1 - K = 10							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.288	0.403	0.4835	0.551	0.598	0.571	0.494
4	0.367	0.355	0.563	0.702	0.732	0.738	0.735
8	0.331	0.354	0.422	0.602	0.695	0.697	0.700
16	0.257	0.328	0.354	0.413	0.534	0.627	0.6344
32	0.223	0.243	0.234	0.288	0.438	0.584	0.609

Table C20: Efficiency values for first version of parallel algorithm, taking K=10

Speedup - Parallel V1 - K = 15							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.571	0.764	0.999	1.111	1.118	1.132	0.994
4	1.433	1.434	2.448	2.825	2.711	2.956	2.958
8	2.687	2.683	2.857	4.890	5.312	5.601	5.676
16	4.858	5.531	5.606	6.812	9.302	9.942	10.523
32	7.226	7.483	7.838	9.847	13.644	18.765	20.008

Table C21: Speed-up values for first version of parallel algorithm, taking K=15

Efficiency - Parallel V1 - K = 15							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.285	0.382	0.499	0.555	0.559	0.566	0.497
4	0.358	0.358	0.612	0.706	0.677	0.739	0.739
8	0.335	0.335	0.357	0.611	0.664	0.700	0.709
16	0.303	0.345	0.350	0.425	0.581	0.621	0.657
32	0.225	0.233	0.244	0.307	0.426	0.586	0.625

Table C22: Efficiency values for first version of parallel algorithm, taking K=15

Speedup - Parallel V1 - K = 20							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.645	0.809	1.087	1.147	1.237	1.179	1.074
4	1.476	1.452	2.033	2.875	3.019	2.174	3.067
8	2.719	2.895	3.666	4.722	5.663	5.734	5.804
16	5.217	4.880	5.767	6.254	9.220	10.186	10.480
32	6.575	7.689	8.058	9.450	15.176	18.538	20.351

Table C23: Speed-up values for first version of parallel algorithm, taking K=20

Efficiency - Parallel V1 - K = 20							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.322	0.404	0.543	0.573	0.618	0.589	0.537
4	0.369	0.363	0.508	0.718	0.754	0.543	0.766
8	0.339	0.361	0.458	0.590	0.707	0.716	0.725
16	0.326	0.305	0.360	0.390	0.576	0.636	0.655
32	0.205	0.240	0.251	0.295	0.474	0.579	0.635

Table C24: Efficiency values for first version of parallel algorithm, taking K=20

Speedup and Efficiency - Parallel Algorithm V2

Speedup - Parallel V2 - K = 5							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.772	0.644	0.890	0.888	0.781	0.719	0.584
4	1.623	1.108	1.869	1.949	1.870	1.458	1.328
8	2.706	2.084	2.819	3.418	3.429	3.150	2.419
16	4.545	3.646	3.391	5.086	5.908	5.105	4.823
32	7.661	6.320	6.456	6.968	9.480	9.721	8.795

Table C25: Speed-up values for second version of parallel algorithm, taking K=5

Efficiency - Parallel V2 - K = 5							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.386	0.322	0.445	0.444	0.390	0.359	0.292
4	0.405	0.277	0.467	0.487	0.467	0.364	0.332
8	0.338	0.260	0.352	0.427	0.428	0.392	0.302
16	0.284	0.227	0.211	0.317	0.369	0.319	0.301
32	0.239	0.197	0.201	0.217	0.296	0.303	0.274

Table C26: Efficiency values for second version of parallel algorithm, taking K=5

Speedup - Parallel V2 - K = 10							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.617	0.687	0.963	0.965	0.939	0.805	0.661
4	1.270	1.235	2.051	2.091	2.041	1.648	1.488
8	2.037	2.342	3.209	3.913	3.893	3.473	2.747
16	3.981	4.014	4.684	5.700	6.593	6.187	5.336
32	5.620	6.878	6.998	10.101	10.096	10.928	10.155

Table C27: Speed-up values for second version of parallel algorithm, taking K=10

Efficiency - Parallel V2 - K = 10							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.308	0.343	0.481	0.482	0.469	0.402	0.330
4	0.317	0.308	0.512	0.522	0.510	0.412	0.372
8	0.254	0.292	0.401	0.489	0.486	0.434	0.343
16	0.248	0.250	0.292	0.356	0.412	0.386	0.333
32	0.175	0.214	0.218	0.315	0.315	0.341	0.317

Table C28: Efficiency values for second version of parallel algorithm, taking K=10

Speedup - Parallel V2 - K = 15							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.605	0.864	0.978	0.987	0.863	0.811	0.664
4	1.264	1.261	1.873	2.115	2.028	1.645	1.557
8	2.160	2.409	3.157	3.719	3.873	3.474	2.807
16	3.909	4.167	4.536	5.948	6.321	5.849	5.486
32	5.695	7.270	7.144	8.739	10.516	10.955	9.941

Table C29: Speed-up values for second version of parallel algorithm, taking K=15

Efficiency - Parallel V2 - K = 15							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.302	0.432	0.489	0.493	0.431	0.405	0.322
4	0.316	0.315	0.468	0.528	0.507	0.411	0.389
8	0.270	0.301	0.394	0.464	0.484	0.434	0.350
16	0.244	0.260	0.283	0.371	0.395	0.365	0.342
32	0.177	0.227	0.223	0.273	0.328	0.342	0.310

Table C30: Efficiency values for second version of parallel algorithm, taking K=15

Speedup - Parallel V2 - K = 20							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.622	0.831	1.052	1.026	0.922	0.843	0.704
4	1.281	1.291	2.094	2.240	2.133	1.721	1.584
8	2.217	2.400	3.509	3.916	3.862	3.624	2.902
16	3.918	4.221	4.592	6.655	6.708	6.532	5.936
32	6.713	7.204	8.151	8.570	11.436	11.859	10.161

Table C31: Speed-up values for second version of parallel algorithm, taking K=20

Efficiency - Parallel V2 - K = 20							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.311	0.415	0.526	0.513	0.461	0.415	0.352
4	0.320	0.322	0.523	0.560	0.533	0.430	0.396
8	0.277	0.300	0.438	0.489	0.482	0.453	0.362
16	0.244	0.263	0.287	0.415	0.419	0.408	0.371
32	0.209	0.225	0.254	0.267	0.357	0.370	0.317

Table C32: Efficiency values for second version of parallel algorithm, taking K=20

Speedup and Efficiency - Parallel Algorithm V3

Speedup - Parallel V3 - K = 5							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.835	0.849	0.991	1.154	1.184	1.202	1.090
4	1.946	1.440	2.456	2.662	2.855	2.907	2.908
8	3.682	2.734	2.895	4.727	5.243	5.548	5.559
16	6.012	5.247	5.316	7.768	9.168	10.068	10.052
32	7.851	6.625	7.229	8.806	12.674	16.326	16.957

Table C33: Speed-up values for third version of parallel algorithm, taking K=5

Efficiency - Parallel V3 - K = 5							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.417	0.424	0.495	0.577	0.592	0.601	0.545
4	0.486	0.360	0.614	0.665	0.713	0.726	0.727
8	0.460	0.341	0.361	0.590	0.655	0.693	0.694
16	0.375	0.327	0.332	0.485	0.573	0.629	0.628
32	0.245	0.207	0.225	0.275	0.396	0.510	0.529

Table C34: Efficiency values for third version of parallel algorithm, taking K=5

Speedup - Parallel V3 - K = 10							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.663	0.786	1.078	1.251	1.264	1.274	1.164
4	1.543	1.509	1.932	2.792	3.021	3.031	3.033
8	2.828	2.874	3.359	4.566	5.433	5.362	5.865
16	5.212	4.750	5.449	7.195	9.432	10.336	10.160
32	7.049	7.346	8.052	10.152	14.267	18.465	20.093

Table C35: Speed-up values for third version of parallel algorithm, taking K=10

Efficiency - Parallel V3 - K = 10							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.331	0.393	0.539	0.625	0.632	0.637	0.582
4	0.357	0.377	0.483	0.698	0.755	0.757	0.758
8	0.353	0.359	0.419	0.570	0.679	0.670	0.731
16	0.325	0.296	0.340	0.449	0.589	0.646	0.635
32	0.220	0.229	0.251	0.317	0.445	0.577	0.627

Table C36: Efficiency values for third version of parallel algorithm, taking K=10

Speedup - Parallel V3 - K = 15							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.643	0.777	1.100	1.221	1.259	1.271	1.165
4	1.495	1.680	2.169	2.782	2.966	3.049	3.071
8	2.023	2.913	3.201	4.980	5.556	5.834	5.896
16	4.699	5.405	5.598	7.444	9.717	9.048	9.239
32	6.991	7.672	7.975	9.309	14.557	18.311	18.287

Table C37: Speed-up values for third version of parallel algorithm, taking K=15

Efficiency - Parallel V3 - K = 15							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.321	0.388	0.55	0.610	0.629	0.635	0.582
4	0.373	0.420	0.542	0.695	0.741	0.762	0.767
8	0.252	0.364	0.400	0.622	0.694	0.729	0.737
16	0.293	0.337	0.349	0.465	0.607	0.5655	0.577
32	0.218	0.239	0.249	0.209	0.454	0.572	0.571

Table C38: Efficiency values for third version of parallel algorithm, taking K=15

Speedup - Parallel V3 - K = 20							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.664	0.943	1.146	1.290	1.308	1.289	1.226
4	1.516	1.526	2.577	2.906	3.072	3.104	3.136
8	2.412	2.826	3.661	4.427	5.756	5.918	5.958
16	4.266	5.545	5.877	8.441	9.771	10.398	10.543
32	6.857	7.408	8.132	9.709	14.512	18.943	20.624

Table C39: Speed-up values for third version of parallel algorithm, taking K=20

Efficiency - Parallel V3 - K = 20							
P/N	1024	2048	4096	8192	16384	32768	65536
2	0.332	0.471	0.573	0.645	0.654	0.644	0.613
4	0.379	0.381	0.644	0.726	0.768	0.776	0.784
8	0.301	0.353	0.457	0.553	0.719	0.739	0.744
16	0.266	0.346	0.367	0.527	0.610	0.649	0.658
32	0.214	0.231	0.254	0.303	0.453	0.591	0.6445

Table C40: Efficiency values for third version of parallel algorithm, taking K=20