Università degli Studi di Camerino

School of Science and Technologies

Course of Master Degree in Computer Science

KNN search MPI
Approaching to KNN search problem for a 3D space using MPI

Group

Yuri Monti

Simone De Bernardinis

Supervisor

Andrea Polini

Contacts

yuri.monti@studenti.unicam.it

simone.debernardinis@studenti.unicam.it

A.Y. 2022/2023

Contents

List of Figures

1. Introduction

In statistics, the k-nearest neighbors algorithm (k-NN) is a non-parametric supervised learning method first developed by Evelyn Fix and Joseph Hodges in 1951, and later expanded by Thomas Cover. It is used for classification and regression. The project focuses to solve the problem, where given points residing on a 3D space it is necessary to identify the K nearest neighbor for each point in the space. The space can be assumed to be an Euclidean space and the usual distance definition can be used. The execution of this project should be done in parallel using multi-core processor, thanks to the usage of MPI (Message Passing Interface) standard.

1.1 Structure of the Document

First of all, this document start describing how the KNN search algorithm works. Regarding chapters 3, 4 and 5, we will outline the approaches we've encountered while addressing the project, defining the versions for both sequential and parallel execution using MPI. Then, in the following chapter 6, we will present comprehensive tables displaying the execution times for all versions, encompassing both parallel and sequential components.

2. General KNN Search Algorithm

In the following discussion, we will focus on the fundamental ideas that form the basis of the algorithms presented in the subsequent chapters. These key concepts serve as the cornerstone upon which the methodologies and techniques to be examined in the following text are built. By exploring these main ideas, we will delve into the heart of the algorithms, understanding the logic and strategies that inform the proposed solutions for a wide range of problems within the specific domain under consideration. Clarity and understanding of these foundations are essential for effectively tackling the algorithmic challenges that will be discussed later on.

2.1 3D Space Distance

In our study, we work with a three-dimensional space and calculate distances between points using the Euclidean formula. This formula, based on geometry principles, allows us to precisely measure distances in this spatial context.
By applying this straightforward approach, we lay a solid groundwork for developing algorithms that effectively handle spatial data and relationships. Given the two points (x_1, y_1, z_1) and (x_2, y_2, z_2), the Euclidean distance is expressed as:

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

2.2 Data Structures

As previously discussed, the data in this problem consists of points that exist within a 3D space, so each point is characterized by three coordinates. For this reason and also for the utility provided by MPI that supports the creation of struct types (Code 2.2), we decided to use a struct that represent a Point in that context (Code 2.1).

Code 2.1: Point Struct

```
typedef struct point_st {
   double x;
   double y;
   double z;
}t_point;
```

Code 2.2: MPI Point Struct

```
MPI_Init
...
MPI_Datatype point_type;
int block_length[] = {1,1,1};
MPI_Aint displacements[] = {0, sizeof(double), 2*sizeof(double)};
MPI_Datatype types[3] = { MPI_DOUBLE, MPI_DOUBLE, MPI_DOUBLE };
MPI_Type_create_struct(3,block_length,displacements,types,
   &point_type);
MPI_Type_commit(&point_type);
...
MPI_Finalize
```

This allowed us to manage the code and positions on arrays and matrices more easily, instead of using 3 doubles and using offsets. The task involves generating a matrix in which each row corresponds to the K neighbors for a specific point in the space.
So we needed to build:

- **A matrix for the distances** where for each point the distances between all other points in space will be calculated.
- **A matrix for the K minimum distances** where for each point the minimum distances between K points in the space will be calculated.
- **A matrix for the K nearest neighbors** where for each point the K points in the space having the nearest distances will be calculated.

Each entry in the K minimum distance matrix corresponds to the index of the neighbor in the K nearest neighbors matrix at the corresponding position.
In our implementations, as we will see in the next chapters, the distance matrix will be computed only in the first version of the algorithm (Chapter 3), because in the other ones we have considered different approaches.

2.3 Points Generation

Data is generated using random values every time the program runs.
The function 2.3 guarantees that the generated points don't obtain coordinates out of a cube with side of 100.
This limit was a part of the requirements for this project so we decided to hard code this parameter, but if needed it can be easily replaced with an input.

Code 2.3: Points Generation

```
void generate_points(t_point* points, int num_points, int
    cube_length) {
    for (int i = 0; i < num_points; i++) {
        points[i].x = (double)rand() / RAND_MAX * cube_length;
        points[i].y = (double)rand() / RAND_MAX * cube_length;
        points[i].z = (double)rand() / RAND_MAX * cube_length;
    }
}
```

To enhance result accuracy across versions, we opted to utilize a common set of points generated using the discussed random algorithm.

2.4 Run Program

The program, as we said, presents different versions and each of them includes a sequential and a parallel executable code. The former is easily compileable with normal *GCC compiler* and executable as a standalone. The latter, instead, needs to be compiled with a different compiler provided by **MPI** standard, called *mpicc* (Code 2.4).

Code 2.4: Compiling mpi project

```
mpicc -o knnmpi main.c -lm
```

Then the executable file produced needs to be run using *mpirun* passing also the number of processors that you want to use for the execution.
Both executables need some parameters to correctly run:

1. **N**: Number of points that will be generated.
2. **K**: Number of neighbors that we want to take into account for generating the minimum distance matrix and the neighbors one (with dimension NxK).

The execution command for the parallel part will then be as shown in Code 2.5.

Code 2.5: running mpi executable

```
mpirun -np <num_procs> ./knnmpi <N> <K>
```

For the evaluation of the performances we added the possibility of running the program loading the points from a file (in our case *load.txt*) in order to avoid different results by the random generation of points.

Code 2.6: running mpi executable with fixed set of points

```
mpirun -np <num_procs> ./knnmpi <N> <K> <filename>
```

3. KNN V1

Initially, our approach involved creating a distance matrix for each point with all others before populating the K nearest neighbors (Knn) matrix for each point. However, this method posed potential memory consumption issues, especially with a large number of points, such as 65536, requiring storage of 65536 * 65536 points in memory.
As a result, we adjusted the filling process for the distance matrix as described below.

3.1 Sequential V1

Following the above considerations, we started reasoning about a proper way to avoid memory issues, and the solution was to apply the filling mechanism not considering the whole matrix but dividing it into blocks.

3.1.1 Distance Matrix

Each of these blocks has a fixed dimension of 256 * 256. This dimension is hard-coded because we set the minimum number of points (N) to 1024, which increases by powers of 2. Consequently, the block size never exceeds N points. The distances will be computed block by block. For simplicity, in Fig. 3.1, we consider the block dimension as N / 2 instead of fixing it to 256.

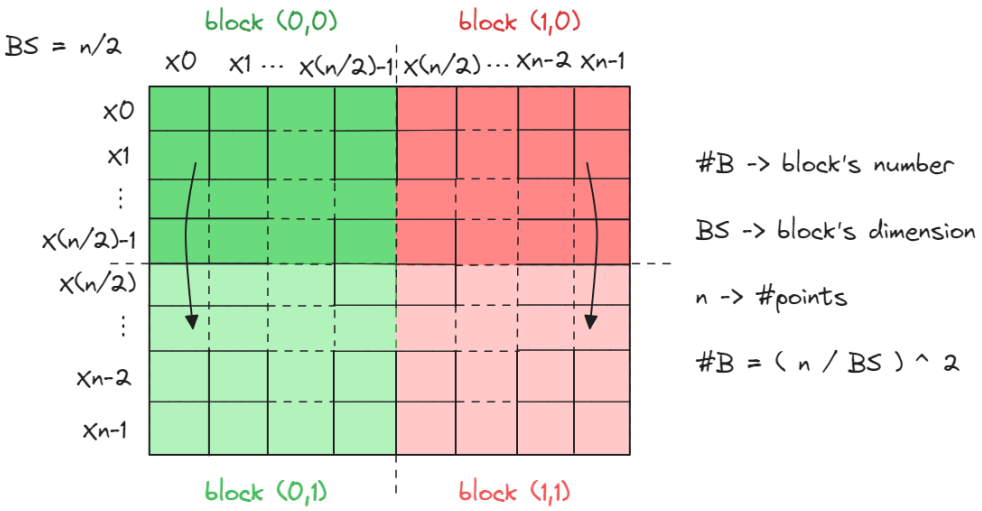

Figure 3.1: Filling distance matrix

3.1.2 Knn Matrices

After completing the filling of each block,the distances are sorted into the K-nearest neighbors (Knn) distance matrix for each of them. Moreover, the indexes of neighbors selected as the K nearest for a current point are stored during the computation. For a graphical representation, please refer to Fig. 3.2.

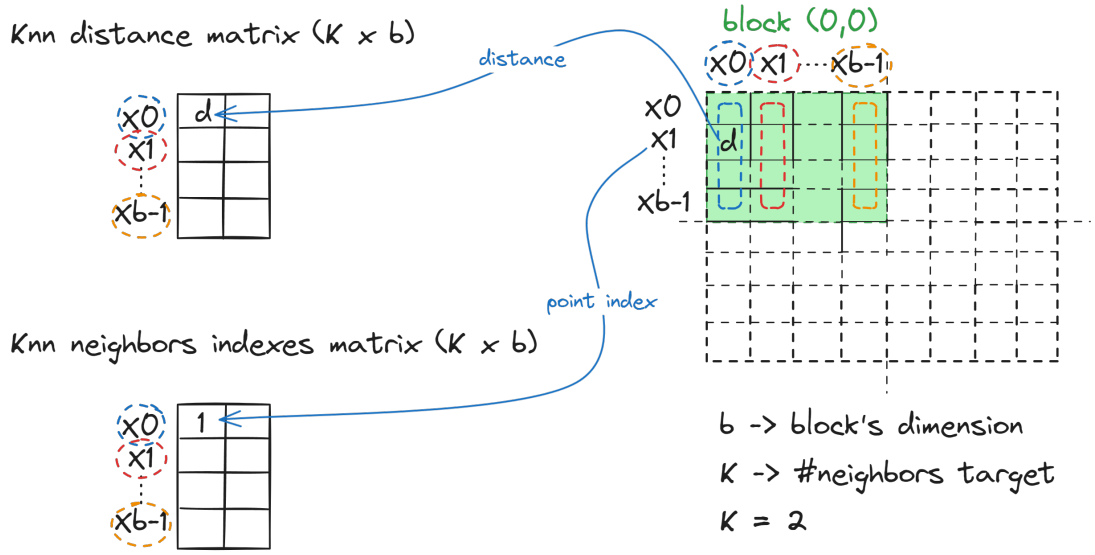

Figure 3.2: Filling Knn matrices

The sorting of the distances for a certain point happens by sliding the Knn matrix as an array from position 0 to K-1, until the considered distance is closer than the Knn one(Code 3.1). This procedure happens for each of the N points.

Code 3.1: Updating Knn matrices

```
void set_values_for_coordinate(int row, int column, int K,double
    *neigh_distances_matrix, int *neighs_matrix, double dist){
   int h; //index of the current position of the knn matrices
   for (h = 0; h < K; h++){
      //pointed distance on knn matrix
      double neigh_dist = neigh_distances_matrix[row*K + h];
      //compare current distance with the pointed one
      if(dist < neigh_dist){
         // right shift from pointed position
         right_shift_from_position(neighs_matrix,
            neigh_distances_matrix,K,h,row);
         //update the pointed knn distance and neighbors index
         set_values_to_neigh(neigh_distances_matrix,neighs_matrix,
            K,dist,row,h,column);
         break;
      }
   }
}
```

3.2 Parallel V1.0

For all parallel versions we used the same functions and algorithm of their sequential counterpart, as far as possible. In this particular version, we didn't use the block dimension as an arbitrary value. Instead, the size is determined based on the number of processors. Nonetheless, the mechanism remains similar in principle. This sub-matrices have dimension:

$$points\ per\ processor * points\ per\ processor$$

where

$$points\ per\ processor = number\ of\ points\ /\ number\ of\ processor$$

It functions similarly, however, in this case, each processor simultaneously takes control of a portion of the matrix. The size of each portion is determined as *points per processor*$*$ N.

3.2.1 Processors Classification

The processors have 2 classifications:

- **Worker processor**: this processor is the unit that works in parallel with others and in our context is responsible for calculating the block distance matrix and sorting those into the Knn matrices. In this class of processors takes part all processor involved into the execution of the program.
- **Coordinator processor**: this processor has the same responsibilities of the Worker one, but it also accounts for the coordination of the program. In our context, this type of processor broadcast and collect data for the computation, composes the complete Knn matrices for the result and is also responsible for the input/output operation like writing the results on a file.

In MPI, the classification of processors is left to the programmer, as it provides access to the processors through their IDs, referred to as **ranks**. We chose the *rank 0* for the Coordinator,and *rank[1..P-1]* (where P is number of processors) for the Worker Processors.

3.2.2 Processors Communication and Computation

In the parallel version, prior to initiating distance calculations, we must setup the data for each processor. To do so, the points need to be distributed to all processors in order to start the computation. This distribution mechanism is implemented using a macro provided by MPI, known as *MPI Bcast* (Code 3.2).

Code 3.2: Points Broadcasting

```
//processor with rank 0 broadcasts to the others processors
//points data, via the Common COMMUNICATOR
MPI_Bcast(points,N,point_type,0,MPI_COMM_WORLD);
```

Following the data distribution, each processor computes the distances for its assigned sub-matrix using a similar approach to the sequential version (as described in subsection 3.1.1). However, in the parallel version, special attention is given to the indexes, as

specific sub-matrices need to be processed for each processor. To do so, we take into account the processor rank (refer to Code 3.3).

Code 3.3: Filling Distance Matrix Parallel V1.0

```
int b_row,i,j;
   for(b_row = 0; b_row<num_procs; b_row++){
      for (i = my_rank*points_per_process;
       i < (my_rank+1)*points_per_process; i++){
         for (j = b_row*points_per_process;
          j <(b_row+1)*points_per_process; j++){
            double dist;
            dist = euclideanDistance(&points[i],&points[j]);
            distances[((i%points_per_process)*points_per_process)
             +j%points_per_process)] = dist;
         }
      }
      ...
   }
...
```

The construction of the Knn matrices follows the same concept of what was described in the sequential part, working with the sub-matrices previously filled by each processor.

Code 3.4: Filling Knn Matrices Parallel V1.0

```
int b_row,i,j;
   for(b_row = 0; b_row<num_procs; b_row++){
      ...//Filling Distance Matrix described above
      for (i = 0; i < points_per_process; i++){
         for (j = 0; j < points_per_process; j++){
            if(i == j && b_row == my_rank) continue;
            double dist = distances[(i*points_per_process)+j];
            set_values_for_coordinate(i,points_per_process*b_row+j,
             K,neigh_distances_matrix,neighs_matrix,dist);
         }
      }
   }
...
```

When all processors have computed the knn matrices for their portion of the matrix, they have to send the results to the *Coordinator processor* (rank 0), that will be the responsible for collecting and ordering the final data. We implement this mechanism through a function provided by MPI called *MPI Gather* (code 3.5).

Code 3.5: MPI Gather V1.0

```
//Gather for knn distances
MPI_Gather(neigh_distances_matrix,K*points_per_process,
 MPI_DOUBLE,neigh_distances_matrix_buffer,K*points_per_process
 MPI_DOUBLE,0,MPI_COMM_WORLD);
//Gather for neighbors indexes
MPI_Gather(neighs_matrix,K*points_per_process,MPI_INT,
 neighs_matrix_buffer,K*points_per_process,
 MPI_INT,0,MPI_COMM_WORLD);
```

3.3 Parallel V1.1

In the parallel V1.0 the exchange of messages was allowed between the *Coordinator processor* and others *Worker processors* using *Collectives operations* by MPI.
In this version of the parallel one, we have communication between all the processes. In the context outlined in subsection 3.2.2, the Coordinator initiates communication by broadcasting messages to the other Working nodes using MPI's *MPI Bcast* operation. Subsequently, the Coordinator gathers results from the Workers through MPI's *MPI Gather*. The communication infrastructure of the previous algorithm results to be a **Star topology**, in which a central node can communicate to all the others, while a peripheral node can communicate only with the central one.
V1.1 and V1.1-ring, as we will describe in in the following sections, utilize distinct topology infrastructures that allow processes to communicate with each other without relying on a central node. This design is implemented to achieve smaller data transfers compared to the previous version. These topology infrastructures are respectively **Fully Connected** for V1.1 and **Ring Topology** for the V1.1-ring.

3.3.1 Fully Connected Topology

In this type of topology each process can communicate directly with *number of processes - 1* processes, so the complete communication channels result to be equals to *number of processes*. Each process hold in memory, aside from the KNN matrices, two arrays of points: one for its associated points, and another for the received ones.

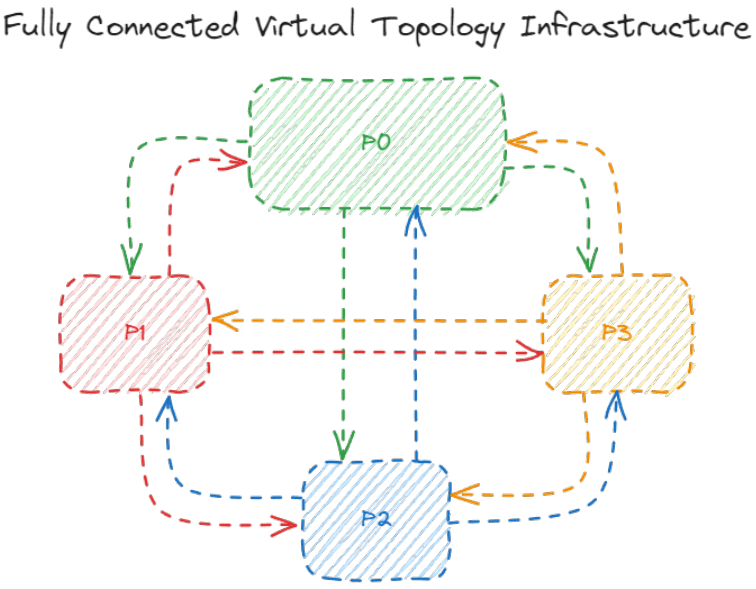

Figure 3.3: Fully Connected Topology with 4 processes V1.1

In our case the communication start with the *Communicator process* (P0 in the figure 3.3) that sends portions of points equal to *number of points / the number of processes* to each process (Code 3.6). We implement this mechanism through functions provided by MPI called:

- *MPI Send* that is used for sending messages.
- *MPI Recv* that is used for receiving messages.

Note that MPI Send is a blocking operation and its behavior defaults to different modes depending on the size of the data it handles:

- Like *MPI Bsend* with a restricted buffer if the data sent hasn't huge dimension; this send is blocked until the buffer is not copied in memory.
- Like *MPI Ssend* when the data sent has huge dimension; this send is blocked until the message hasn't been reached by the recipient that calls Receive.

Code 3.6: Initialization of points into processes V1.1

```
if (my_rank == 0)
   {
      int i,j;
      for (j = 0; j < points_per_process; j++)
      {
         my_points[j] = points[j];
         received_points[j] = points[j];
      }
      for (i = 1; i < num_procs; i++)
      {
         MPI_Send(points + (i * points_per_process),
            points_per_process, point_type, i, 0, MPI_COMM_WORLD);
         MPI_Send(points + (i * points_per_process),
            points_per_process, point_type, i, 1, MPI_COMM_WORLD);
      }
   }
   else
   {
      MPI_Recv(my_points, points_per_process, point_type, 0, 0,
          MPI_COMM_WORLD, &status);
      MPI_Recv(received_points, points_per_process, point_type, 0,
          1, MPI_COMM_WORLD, &status);
   }
```

Now, processors can compute the distance matrix using their own points for the first iteration, similar to what occurred in the parallel version 1.0 (3.2.2). After the KNN values are inserted into KNN matrices, each process begins to exchange its fixed points with others, which are received in the *received_points* buffer, as demonstrated in code listing 3.7.
In this case, we utilized MPI Bsend directly because otherwise, given the large data being exchanged, MPI Send would default to using the Synchronous one (MPI Ssend). This could potentially lead to a deadlock, as we're in a scenario where a Send and Receive occur consecutively.

Code 3.7: Exchange of received points among processes V1.1

```
int buffer_attached_size = MPI_BSEND_OVERHEAD +
   sizeof(t_point)*points_per_process;
         char *buffer_attached = (char
            *)malloc(buffer_attached_size);
         MPI_Buffer_attach(buffer_attached, buffer_attached_size);

         MPI_Bsend(my_points, points_per_process, point_type,
            (my_rank + i) % num_procs, 1, MPI_COMM_WORLD);
```

```
        MPI_Recv(received_points, points_per_process, point_type,
            (my_rank - i + num_procs) % (num_procs), 1,
            MPI_COMM_WORLD,MPI_STATUS_IGNORE);

        MPI_Buffer_detach(&buffer_attached, &buffer_attached_size);
        free(buffer_attached);
```

Once all processors complete the computation of the knn matrices, a *MPI Gather* happens for collecting resulting data by *Coordinator process*, like the V1.0 algorithm (3.5).

3.3.2 Ring Topology

To minimize the number of communication links, we decided to exchange not the fixed points of a process, but only the received ones. In each iteration, a process sends the points it received from the previous one to the next one, following a circular flow. Therefore, the virtual infrastructure in this version of the parallel algorithm represents what is called a **Ring Topology**.

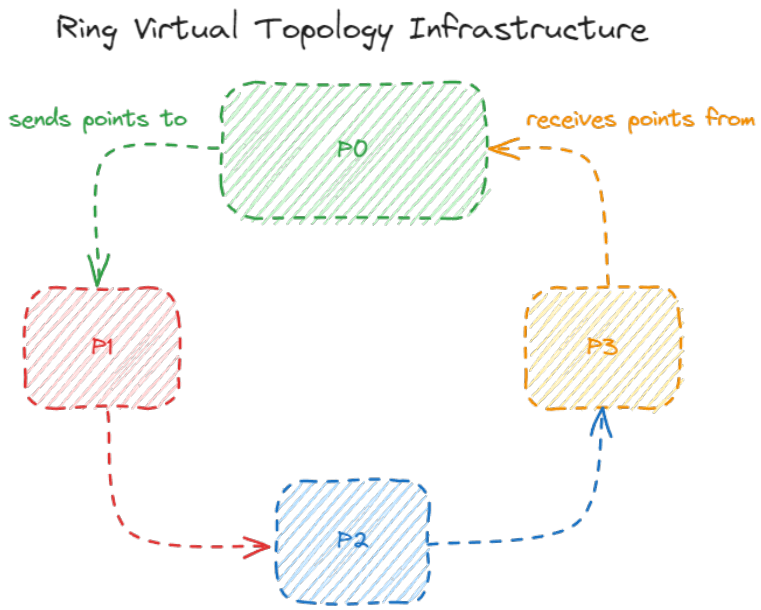

Figure 3.4: Ring Topology with 4 processes V1.1-ring

In this topology, each process can communicate directly only with the next rank process, resulting in a total of *number of processes* communication channels. The behavior is the same of the V1.1, but they differ in the aspect mentioned above(Code 3.8).

Code 3.8: Exchange of received points in a ring V1.1

```
int p = 1;
   while(p < num_procs)
   {
      .
      .
      //A Process sends the received points by the previous process
         to the next one
```

```
        // dest = my_rank + 1 --> next one to the ring
        MPI_Bsend(received_points, points_per_process, point_type,
            dest, RECEIVED_TAG, MPI_COMM_WORLD);
        // source = my_rank - 1 --> previous one to the ring
        MPI_Recv(received_points, points_per_process, point_type,
            source, RECEIVED_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        .
        .
        calculate_and_insert_distance(neigh_distances_matrix,
        neighs_matrix,(my_rank - p +
            num_procs)%(num_procs),points_per_process,K,
        my_points,received_points,false);

        p++;
    }
```

As the V1.1, the computation involved in the distance and Knn matrices are the same of the parallel V1.0 algorithm.

3.4 Results and Performance evaluation

In this section, we will illustrate the speed-up and efficiency achieved by the parallel version compared to the sequential one using charts.
The Speed-up formula is:

$$S(n,p) = \frac{T_{serial}(n)}{T_{parallel}(n,p)} \tag{3.1}$$

The Efficiency formula is:

$$E(n,p) = \frac{S(n,p)}{p} = \frac{T_{serial}(n)}{T_{parallel}(n,p) \times p} \tag{3.2}$$

From the charts and execution time tables in Chapter 6.1, it is evident that the speed-up approaches linearity as the number of involved processors increases.
The efficiency, excluding cases with P = 2 and P = 4, appears to follow weak scalability. Increasing the number of processors while keeping the number of points constant results in decreasing efficiency, and vice versa.

3.4.1 Algorithm 1.0 Performance Charts

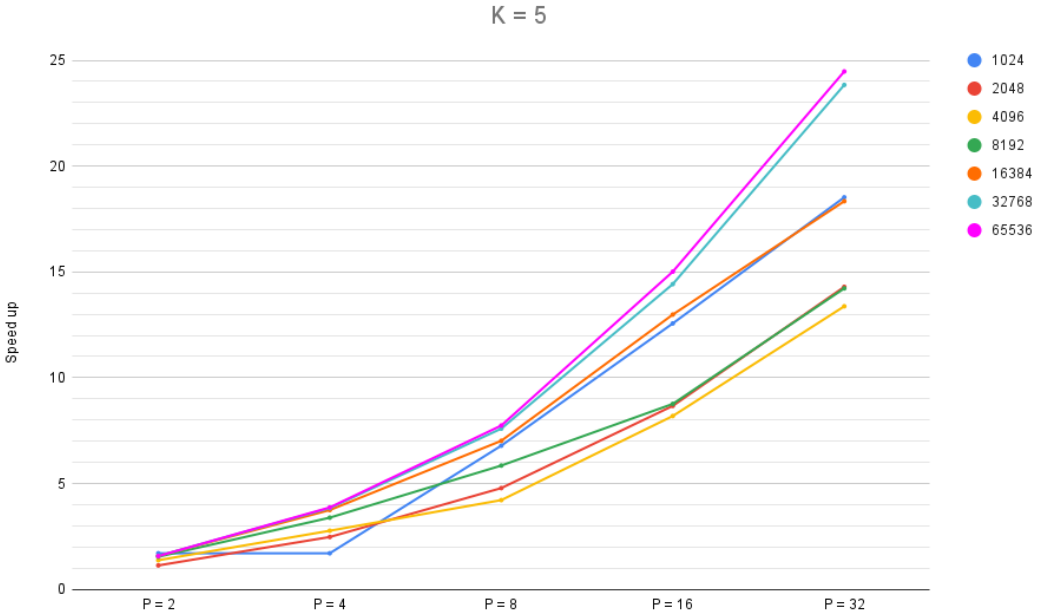

Figure 3.5: Speed-Up V1 with K = 5

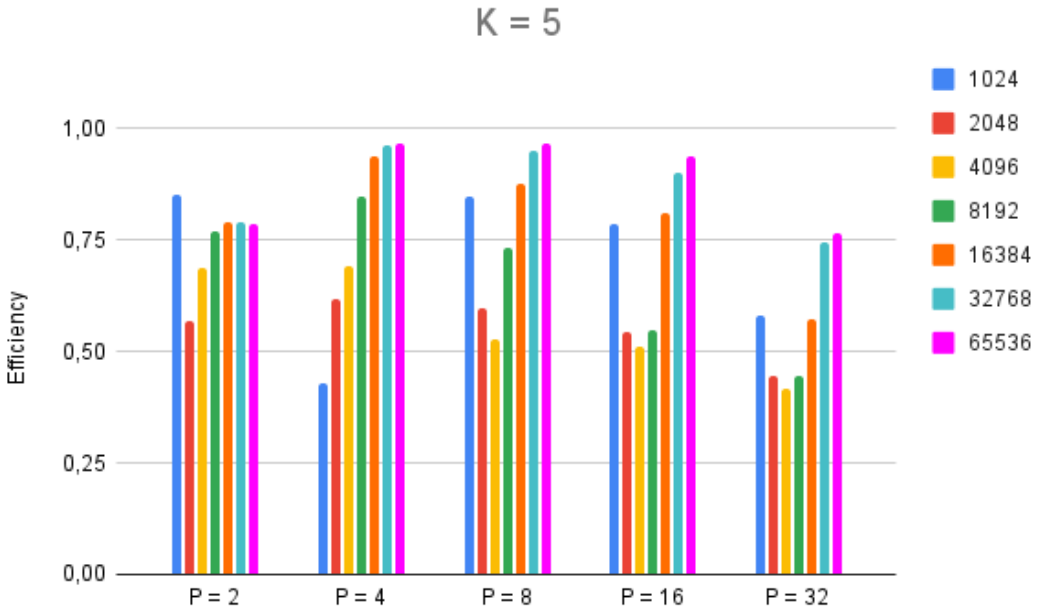

Figure 3.6: Efficiency V1 with K = 5

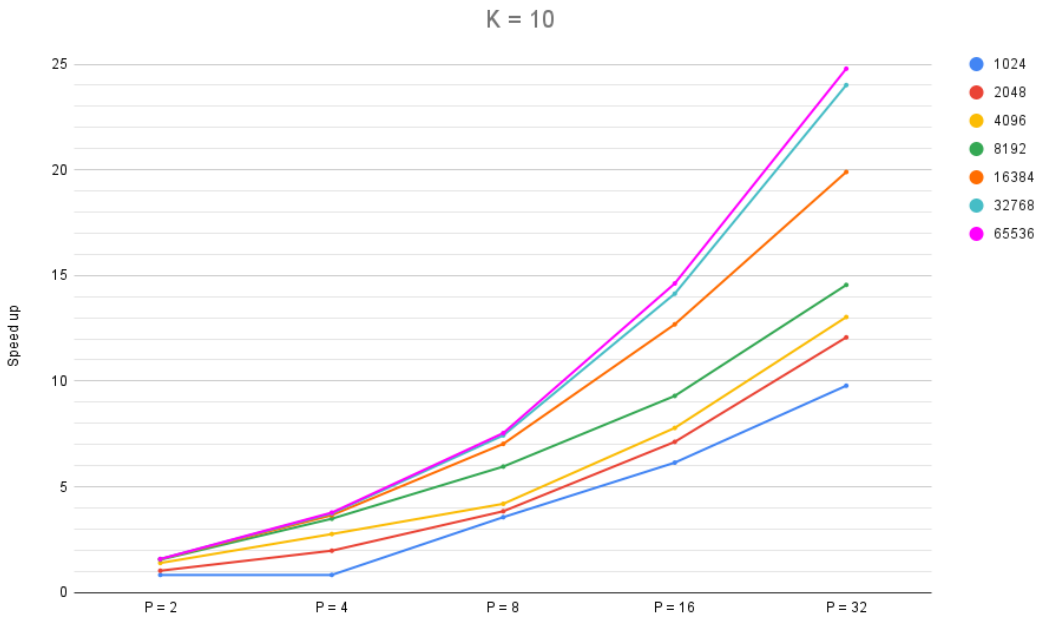

Figure 3.7: Speed-Up V1 with K = 10

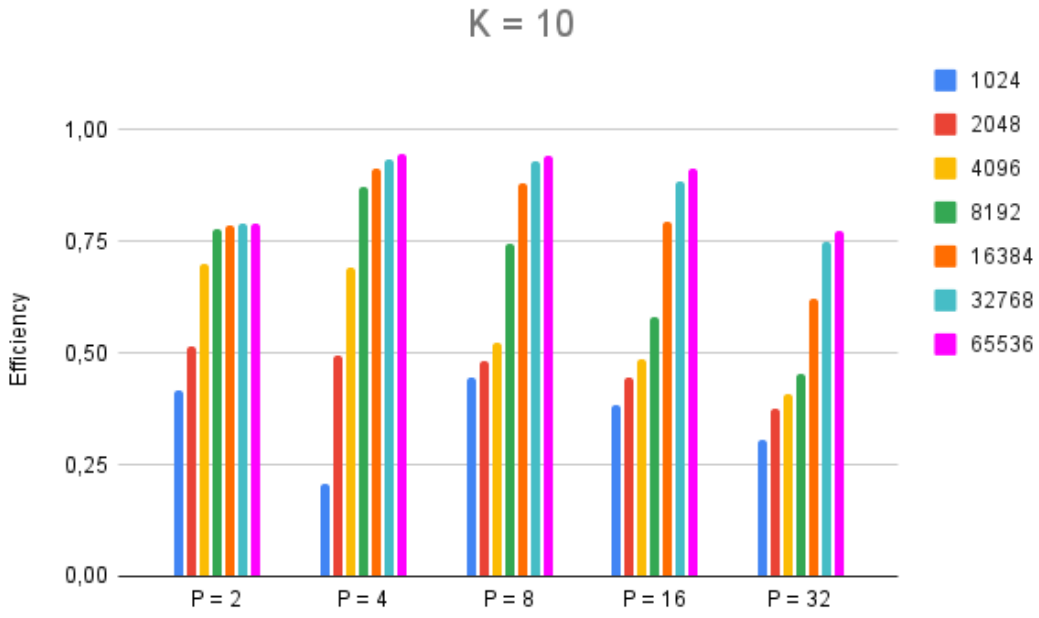

Figure 3.8: Efficiency V1 with K = 10

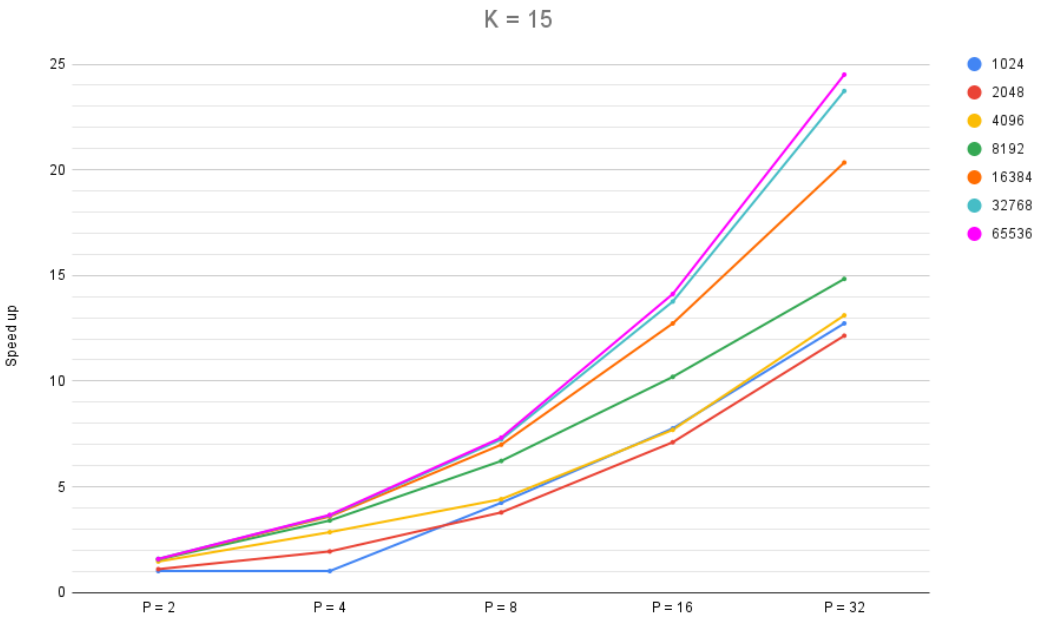

Figure 3.9: Speed-Up V1 with K = 15

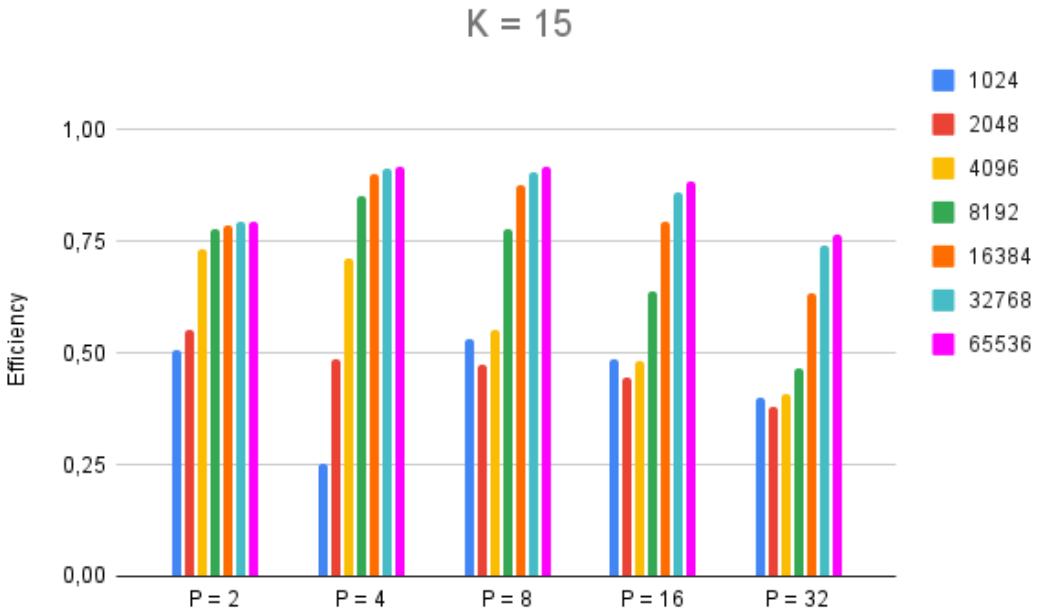

Figure 3.10: Efficiency V1 with K = 15

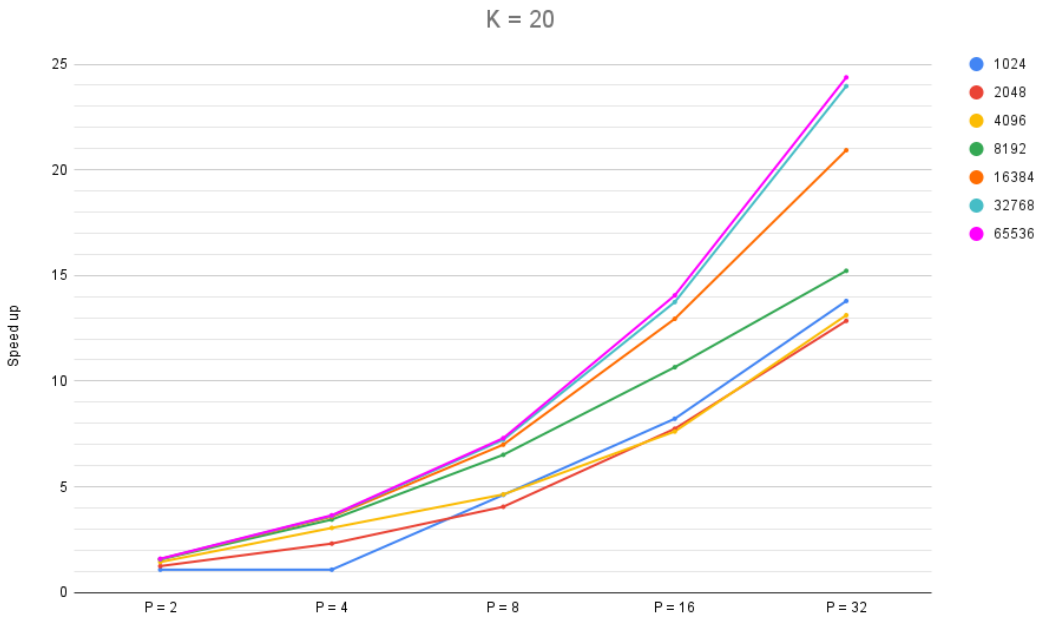

Figure 3.11: Speed-Up V1 with K = 20

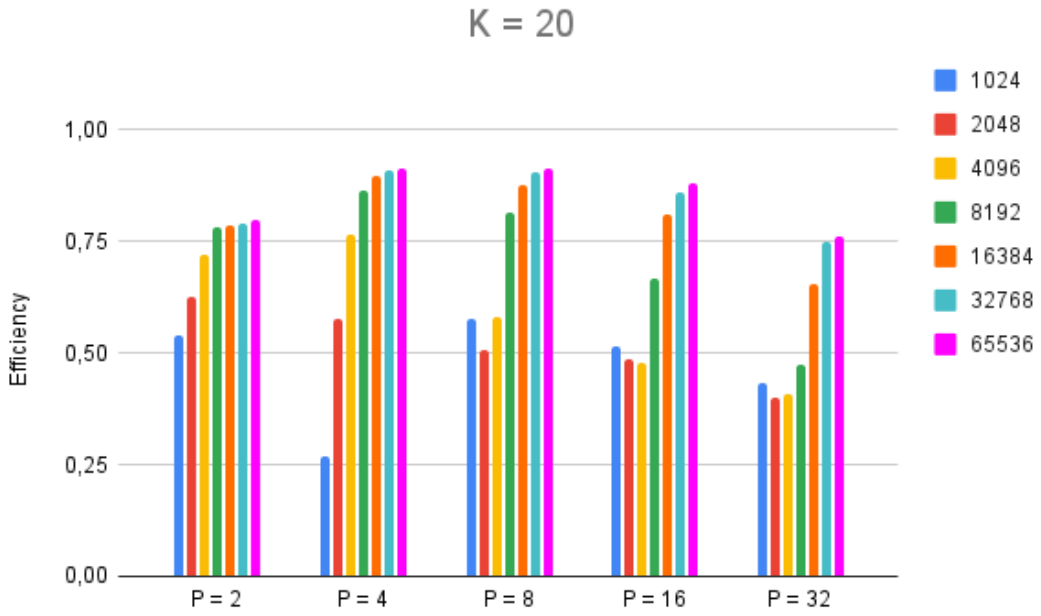

Figure 3.12: Efficiency V1 with K = 20

3.4.2 Algorithm 1.1 Performance Charts

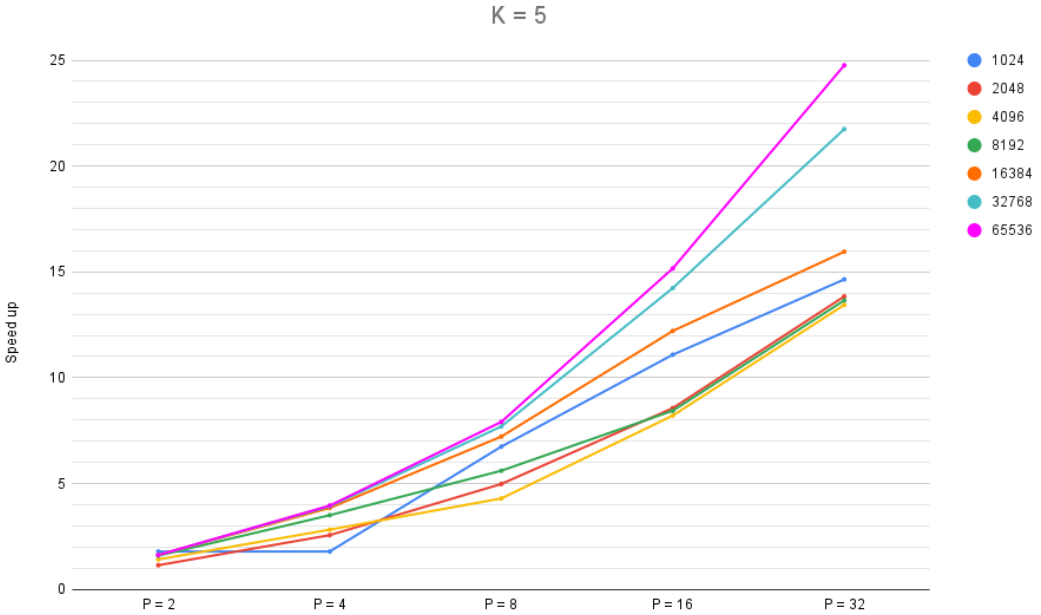

Figure 3.13: Speed-Up V1.1 with K = 5

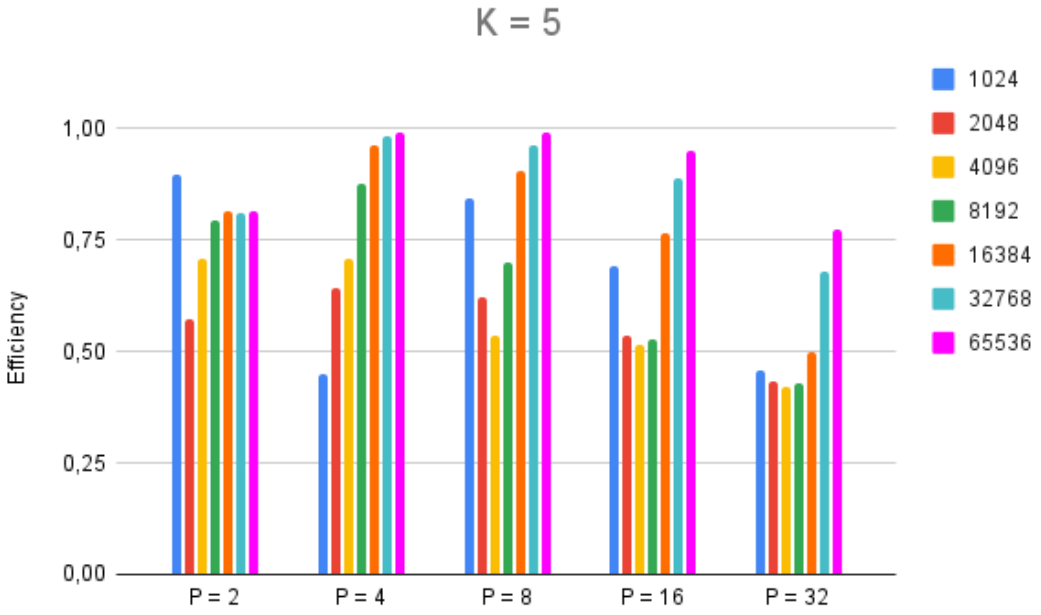

Figure 3.14: Efficiency V1.1 with K = 5

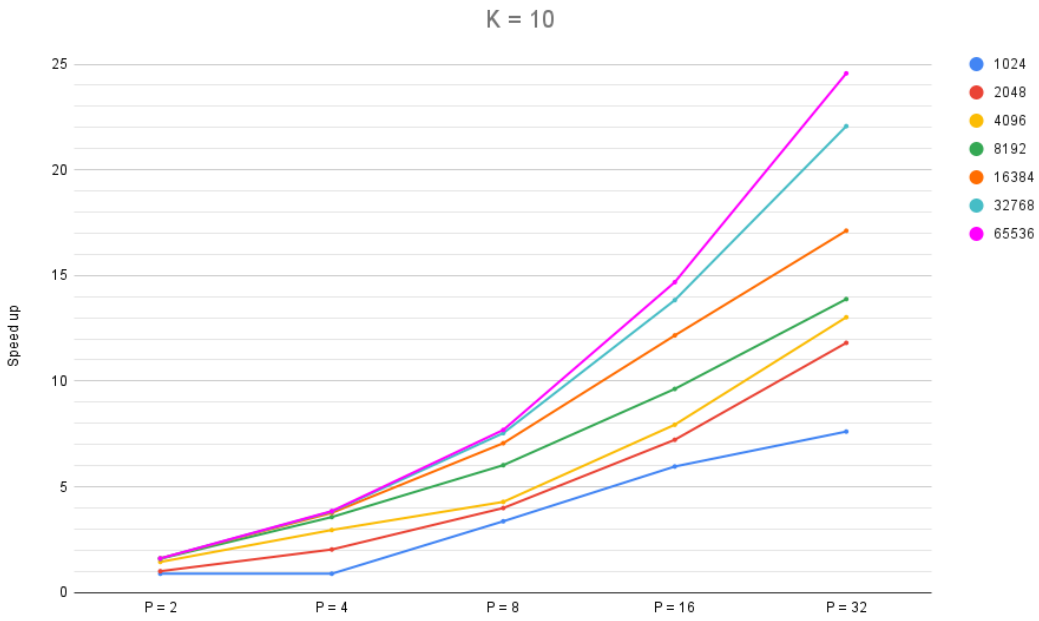

Figure 3.15: Speed-Up V1.1 with K = 10

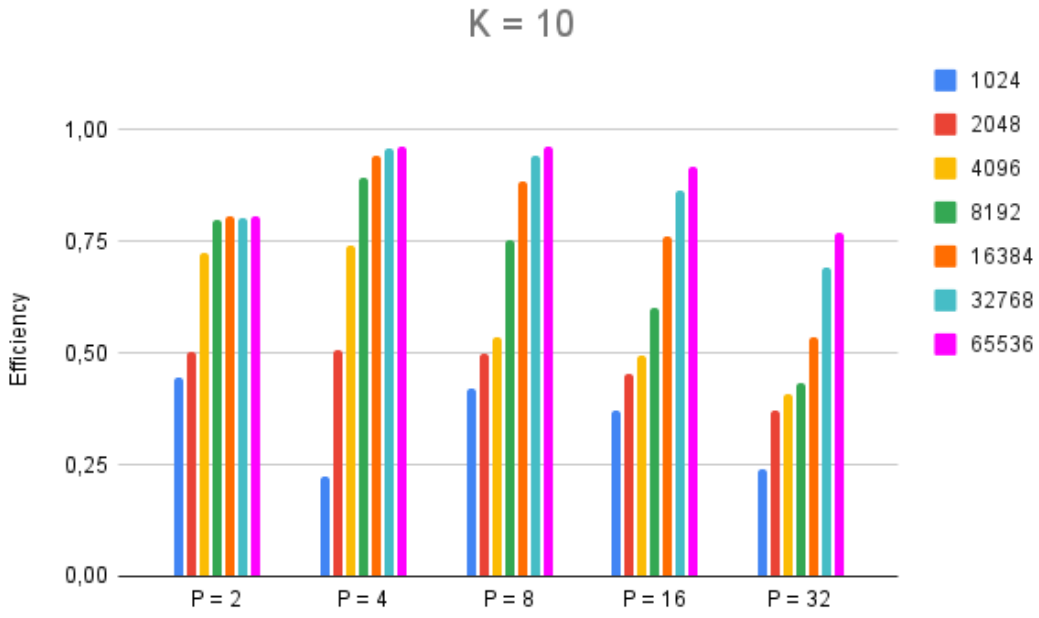

Figure 3.16: Efficiency V1.1 with K = 10

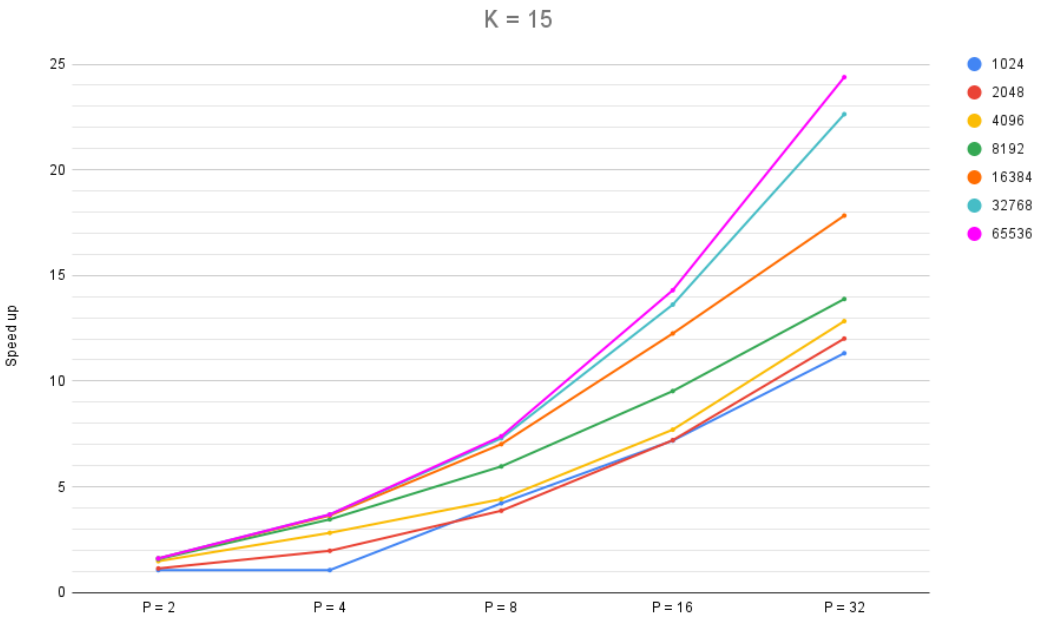

Figure 3.17: Speed-Up V1.1 with K = 15

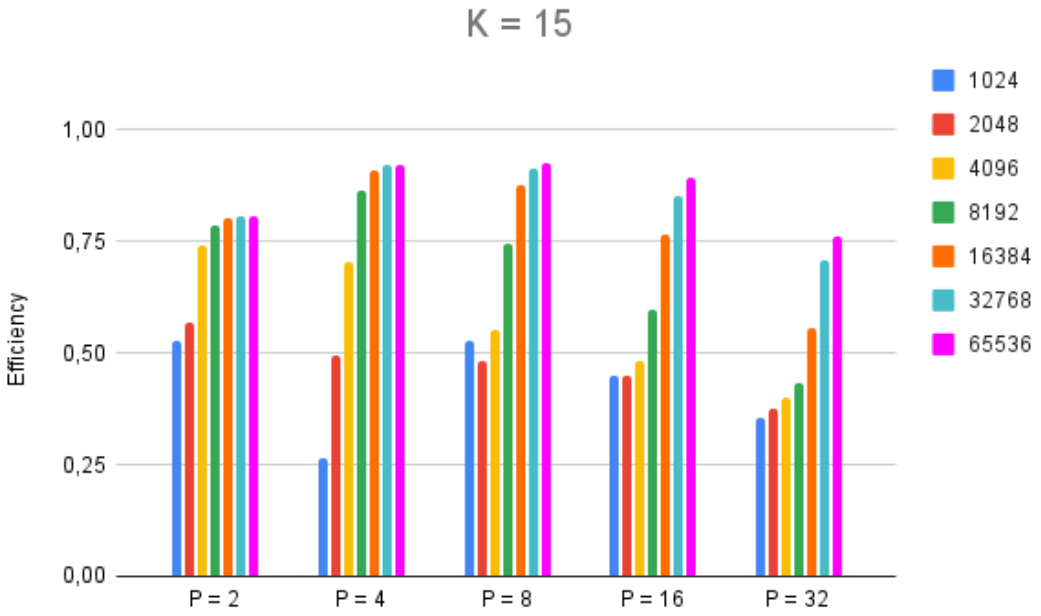

Figure 3.18: Efficiency V1.1 with K = 15

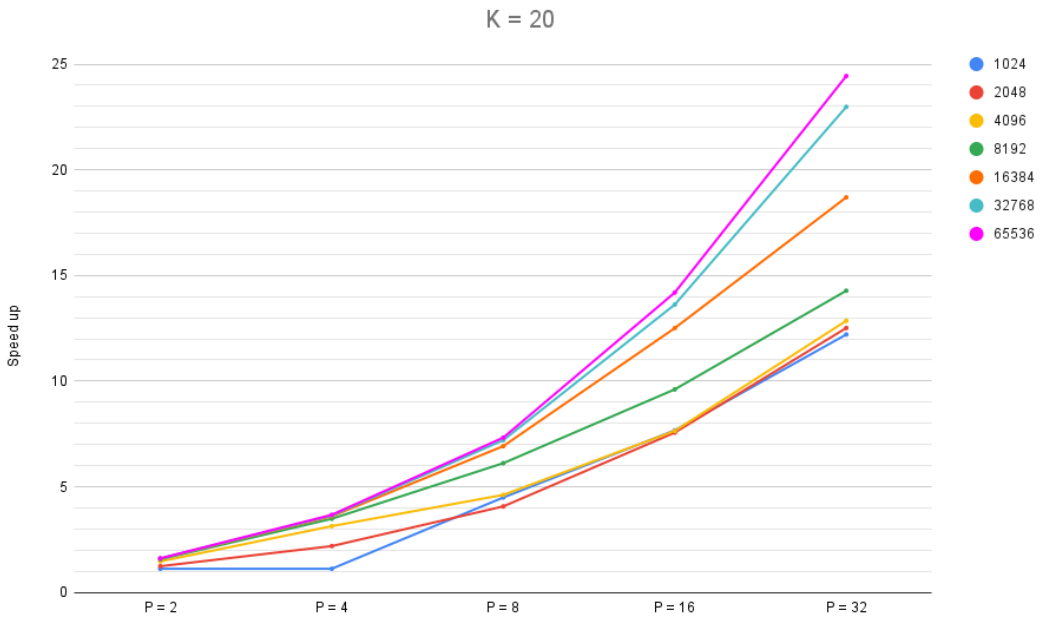

Figure 3.19: Speed-Up V1.1 with K = 20

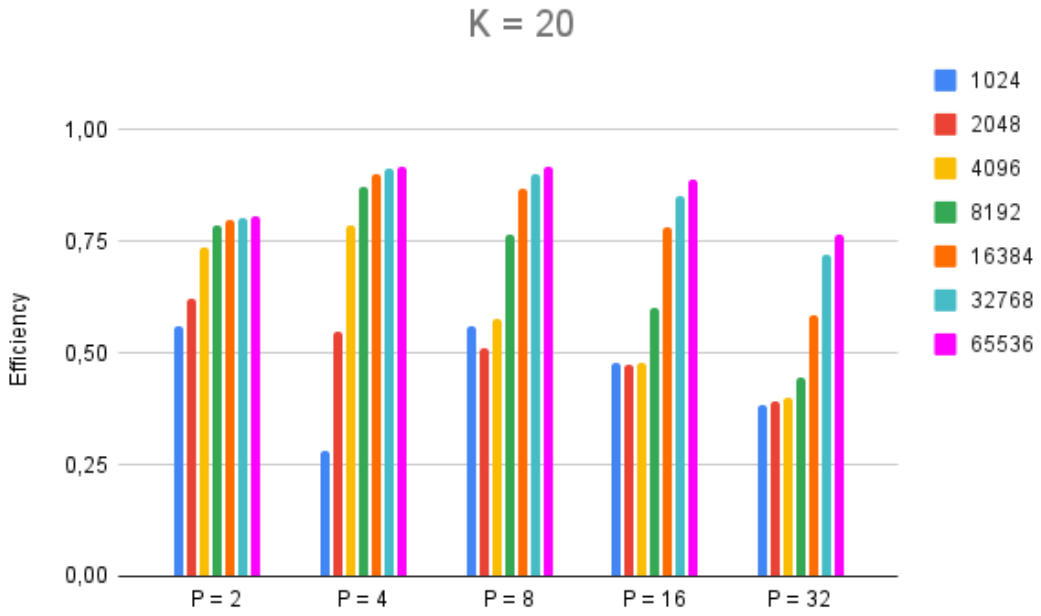

Figure 3.20: Efficiency V1.1 with K = 20

3.4.3 Algorithm 1.1 Ring Performance Charts

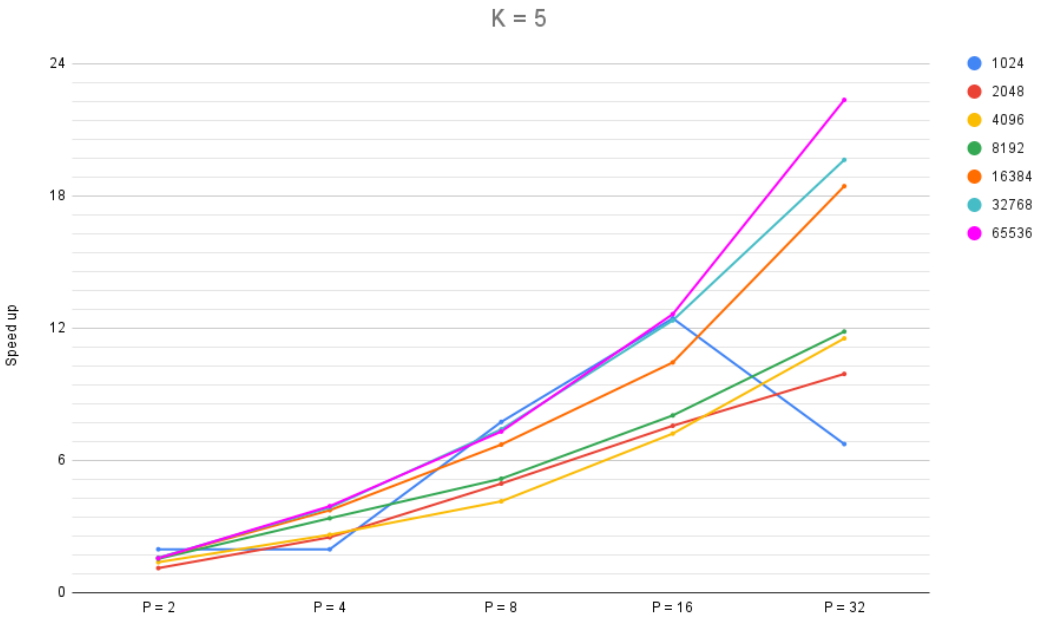

Figure 3.21: Speed-Up V1.1-ring with K = 5

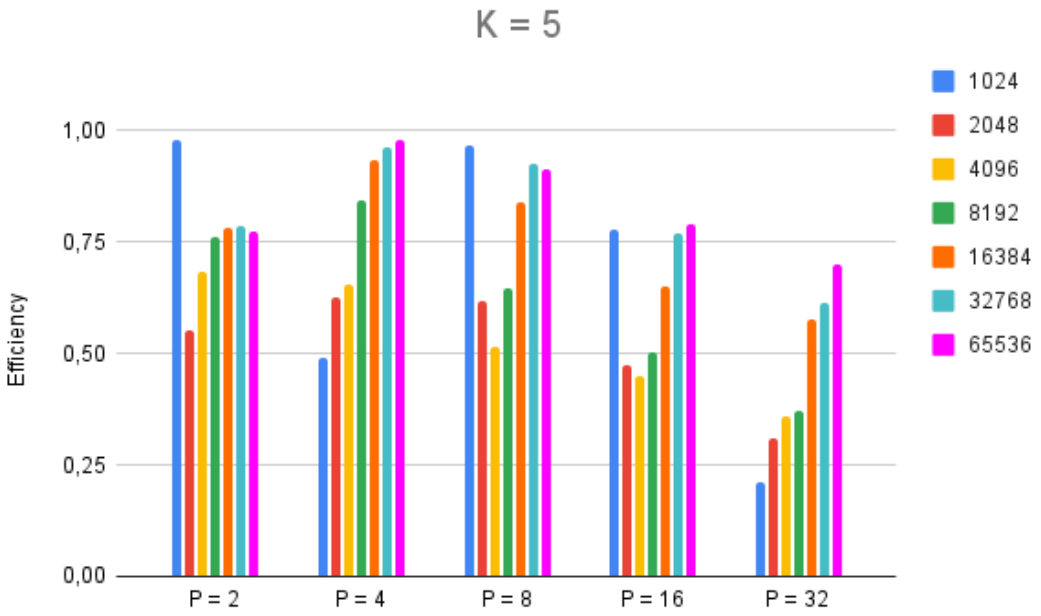

Figure 3.22: Efficiency V1.1-ring with K = 5

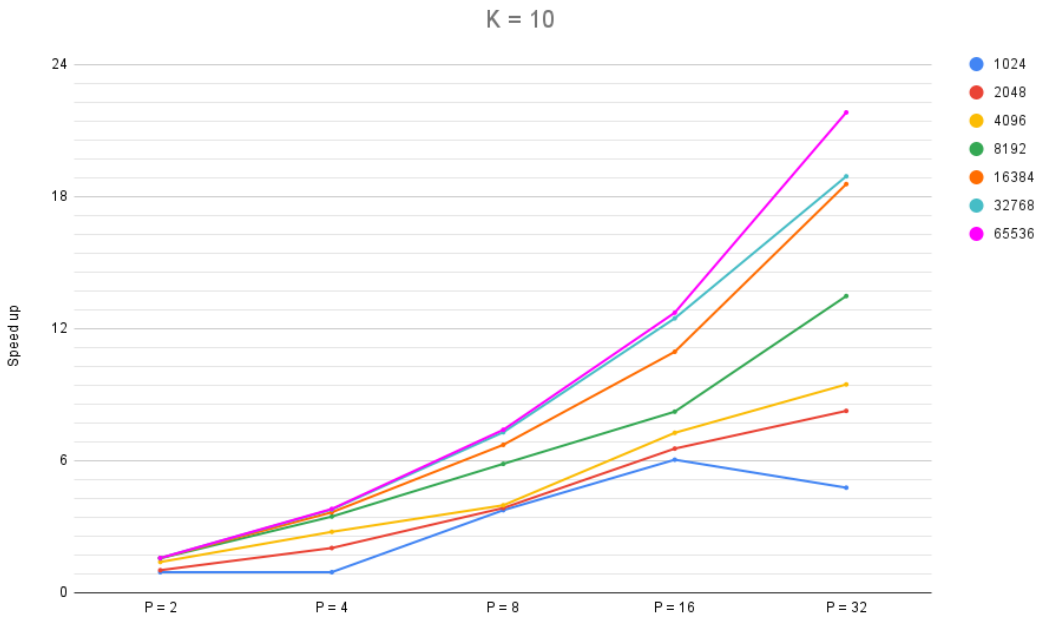

Figure 3.23: Speed-Up V1.1-ring with K = 10

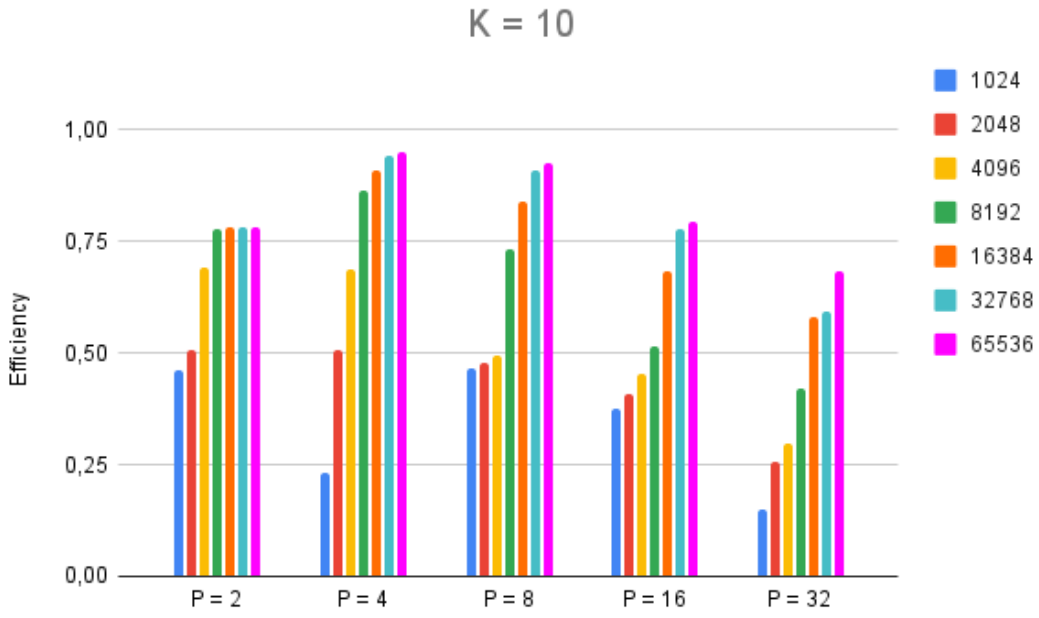

Figure 3.24: Efficiency V1.1-ring with K = 10

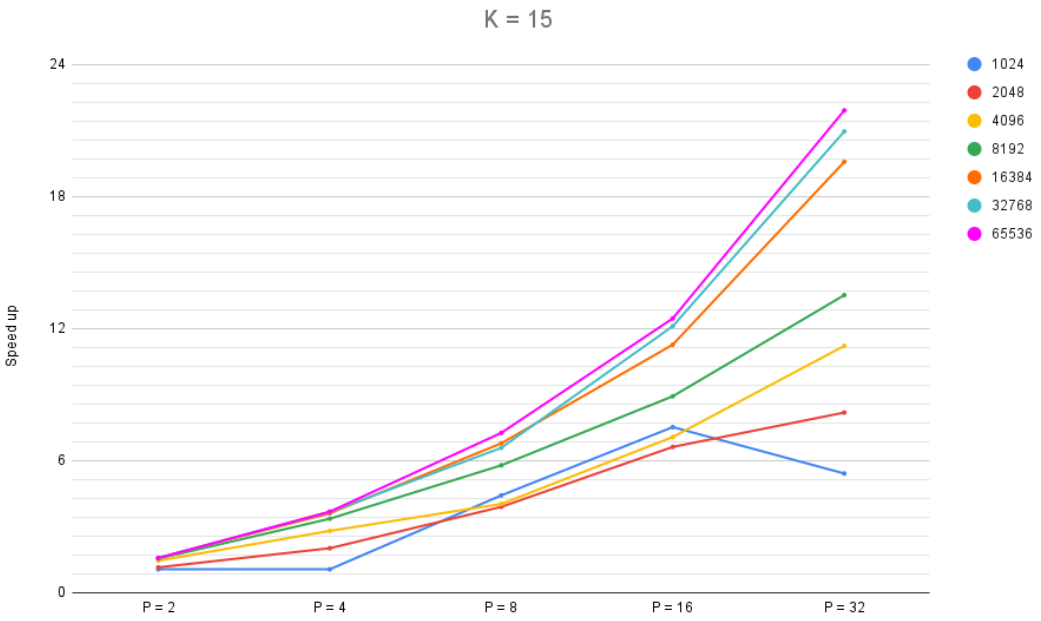

Figure 3.25: Speed-Up V1.1-ring with K = 15

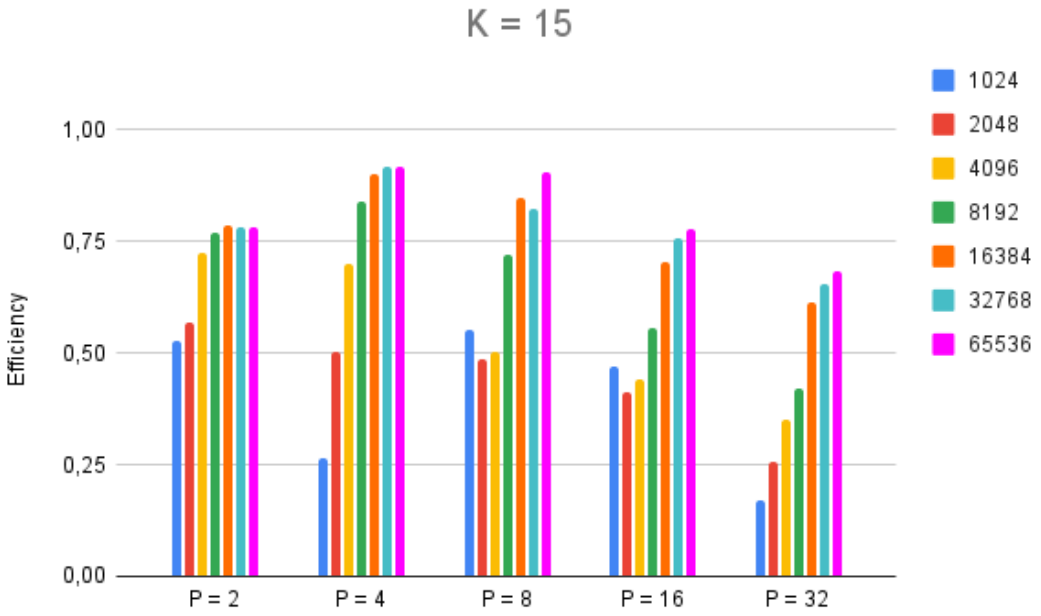

Figure 3.26: Efficiency V1.1-ring with K = 15

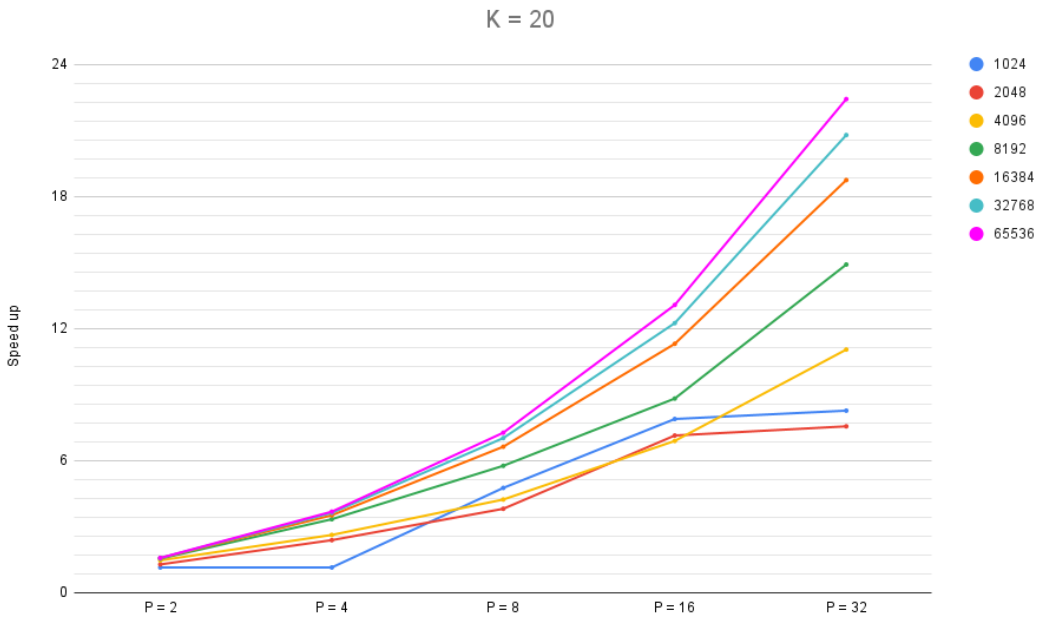

Figure 3.27: Speed-Up V1.1-ring with K = 20

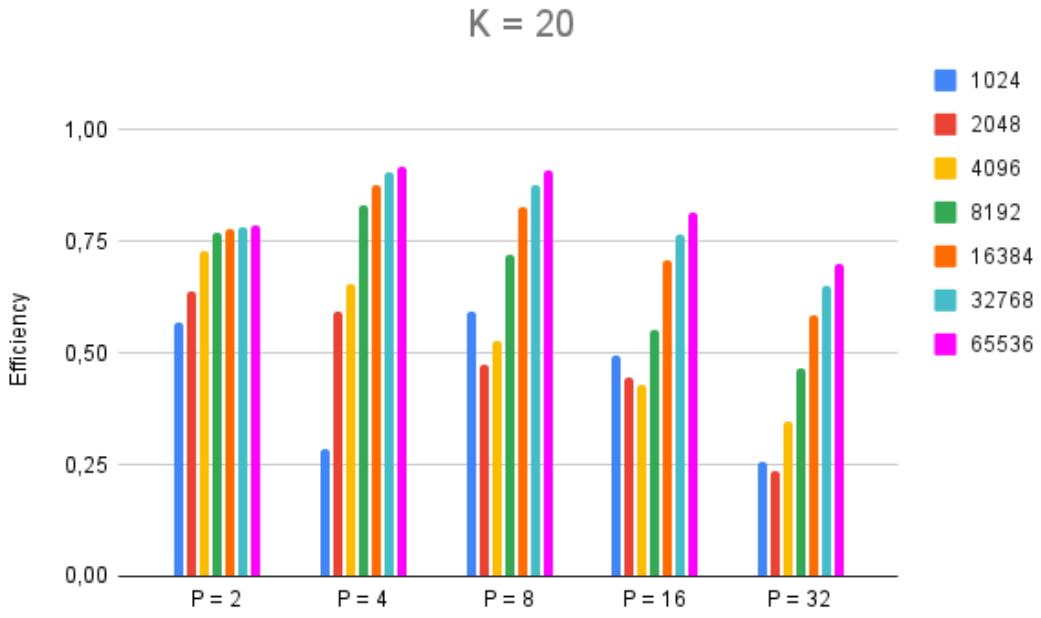

Figure 3.28: Efficiency V1.1-ring with K = 20

4. KNN V2.0

We have decided to alter the approach presented in the first version (Chapter 3) in order to reduce memory allocation. In version 2.0, the block distance matrix is no longer stored in memory. Instead, once the Euclidean distance between two points is computed, it is immediately inserted and sorted into the K-nearest neighbors (Knn) matrices. Consequently, the concept of blocks is abandoned in this version. Despite the decision not to allocate memory for the block matrix, this approach ultimately proved to be less efficient in performance compared to version 1.

4.1 Sequential V2

In this version, as described previously, we calculate the distance point by point and insert it into the K-nearest neighbors (Knn) matrices at the appropriate position using the same sorting method as in the previous version, as illustrated in Fig 4.1. Therefore, it is only necessary to read the points and sort their distances once they are calculated, as shown in Code 4.1.

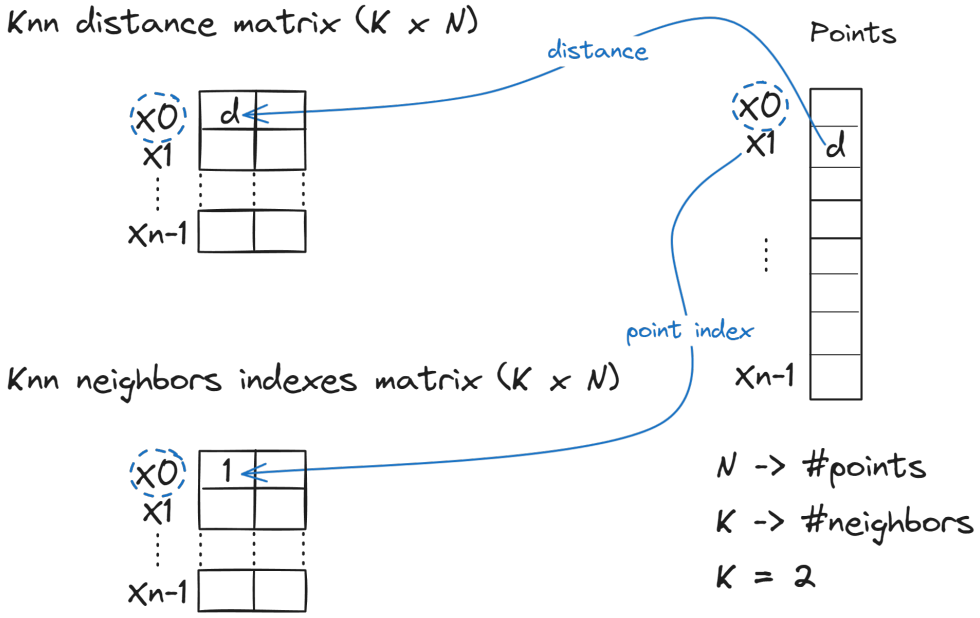

Figure 4.1: Filling Knn matrices V2

Code 4.1: Updating Knn matrices

```
int i,j,h;
```

```
for (i = 0; i < N; i++){
   for (j = 0; j < N; j++){
      if(i == j) continue;
         double dist = euclideanDistance(&points[i],&points[j]);
         for (h = 0; h < K; h++){
            double neigh_dist =
               neigh_distances_matrix[get_matrix_position(i,h,K)];
            if(dist < neigh_dist){
               right_shift_from_position(neighs_matrix,
                 neigh_distances_matrix,K,h,i);
               set_values_to_neigh(neigh_distances_matrix,
                 neighs_matrix,K,dist,i,h,j);
               break;
            }
         }
   }
}
```

4.2 Parallel V2

This parallel version, like the first one, functions similarly to the corresponding sequential version in terms of its underlying functions, but differs in its data handling approach. In this case, the code remains largely the same, although there are operations geared towards processor coordination. The differences primarily lie in the manipulation of indexes during the for iterations. In the Fig 4.1, we show how the parallel version operates with the data.

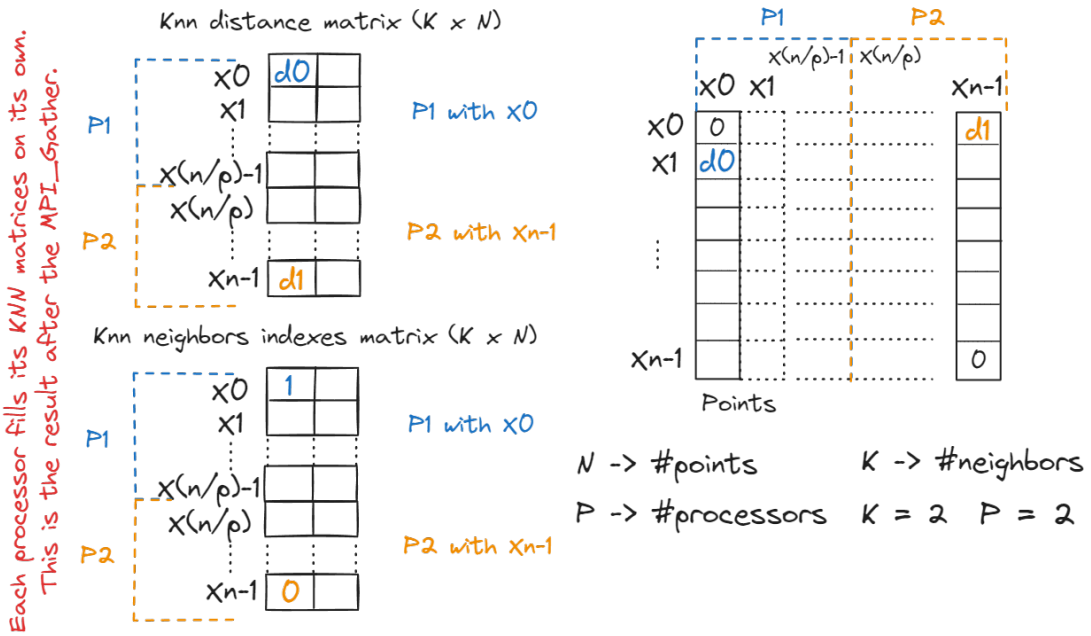

Figure 4.2: Filling Knn matrices Parallel V2

4.3 Results and Performance evaluation

Just as in the previous chapter (Section 3.4), we will showcase the Speedup and Efficiency of this version compared to the sequential one, using the same formulas as before. Below are the charts displaying all recorded values during the execution of both programs. Additionally, in Section 6.4, the execution time tables are provided.

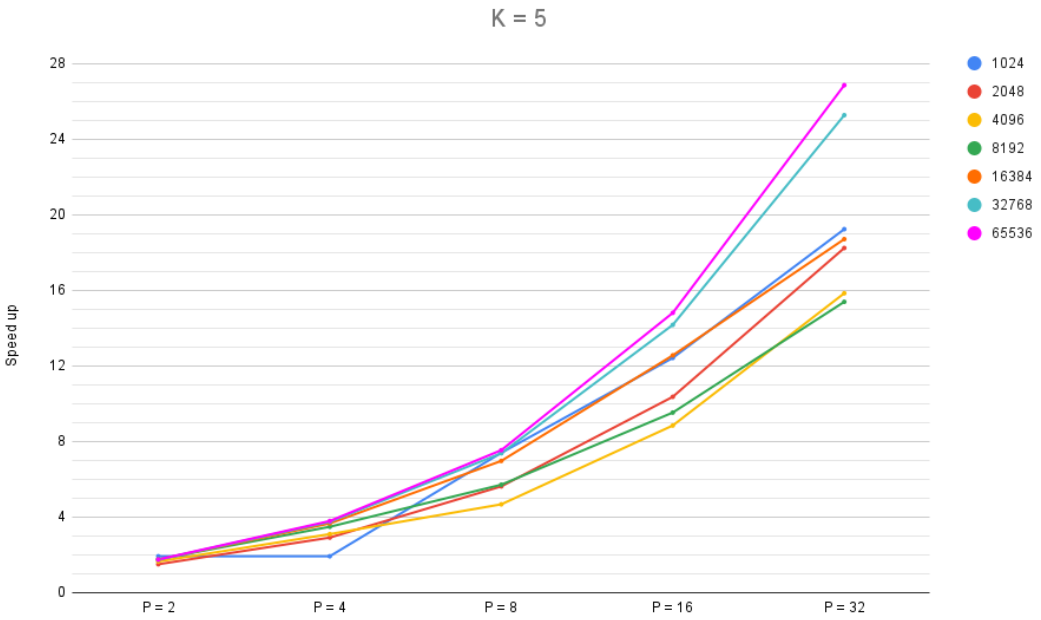

Figure 4.3: Speed-Up V2.0 with K = 5

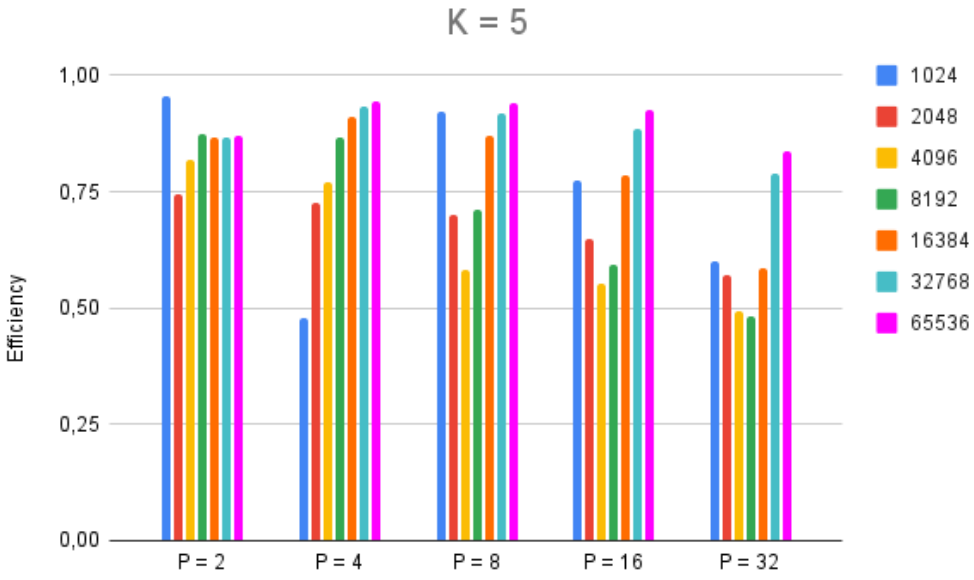

Figure 4.4: Efficiency V2.0 with K = 5

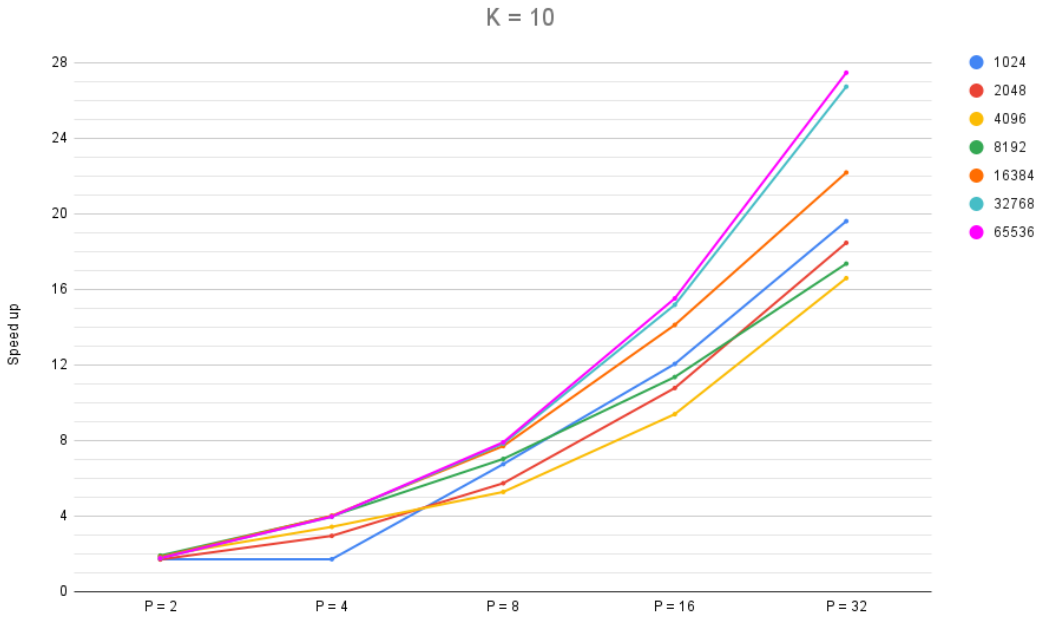

Figure 4.5: Speed-Up V2.0 with K = 10

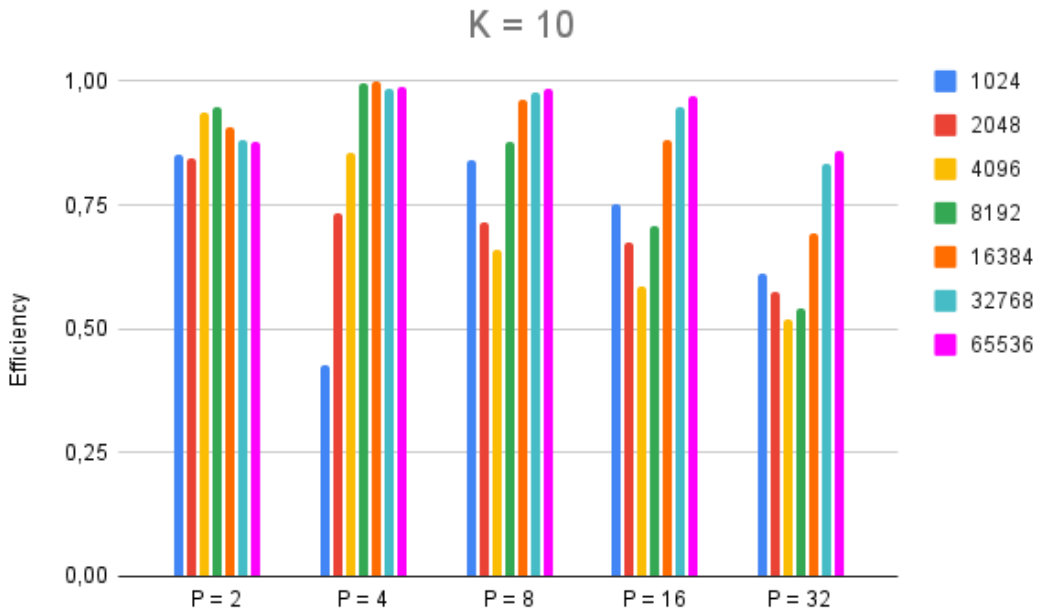

Figure 4.6: Efficiency V2.0 with K = 10

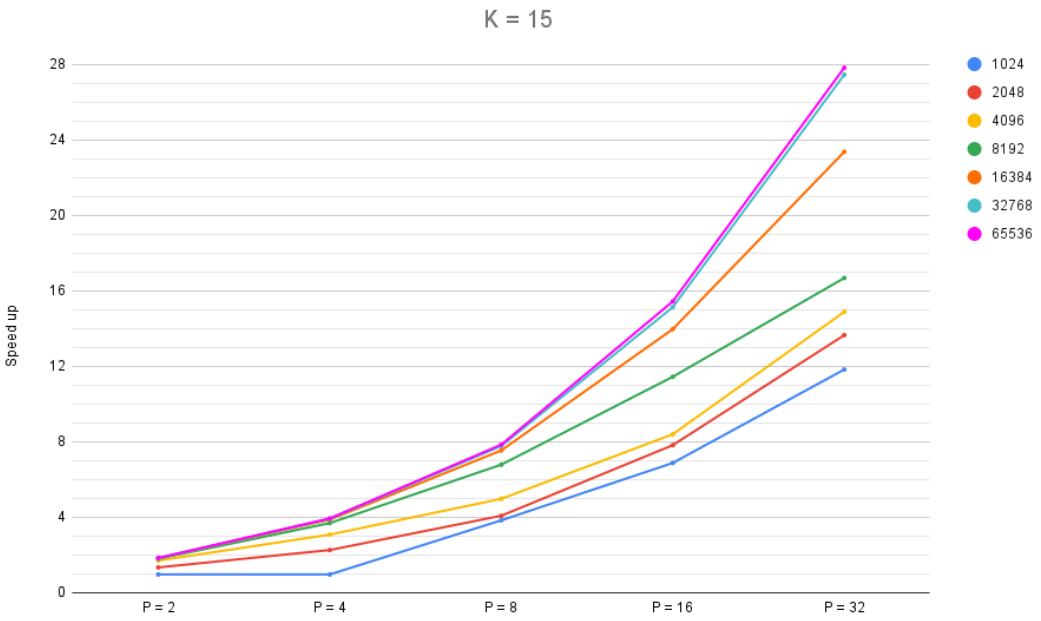

Figure 4.7: Speed-Up V2.0 with K = 15

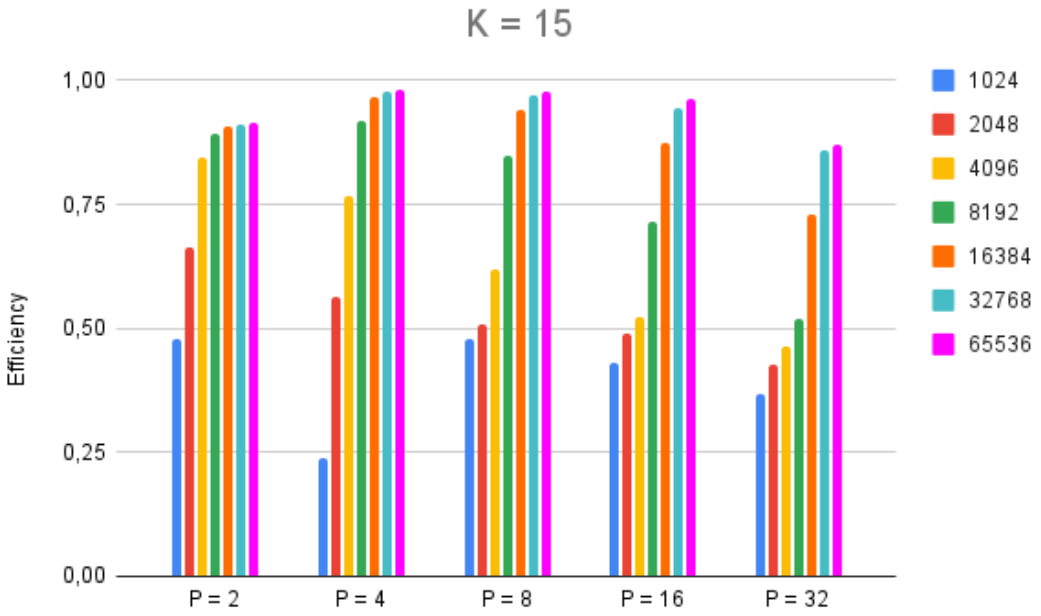

Figure 4.8: Efficiency V2.0 with K = 15

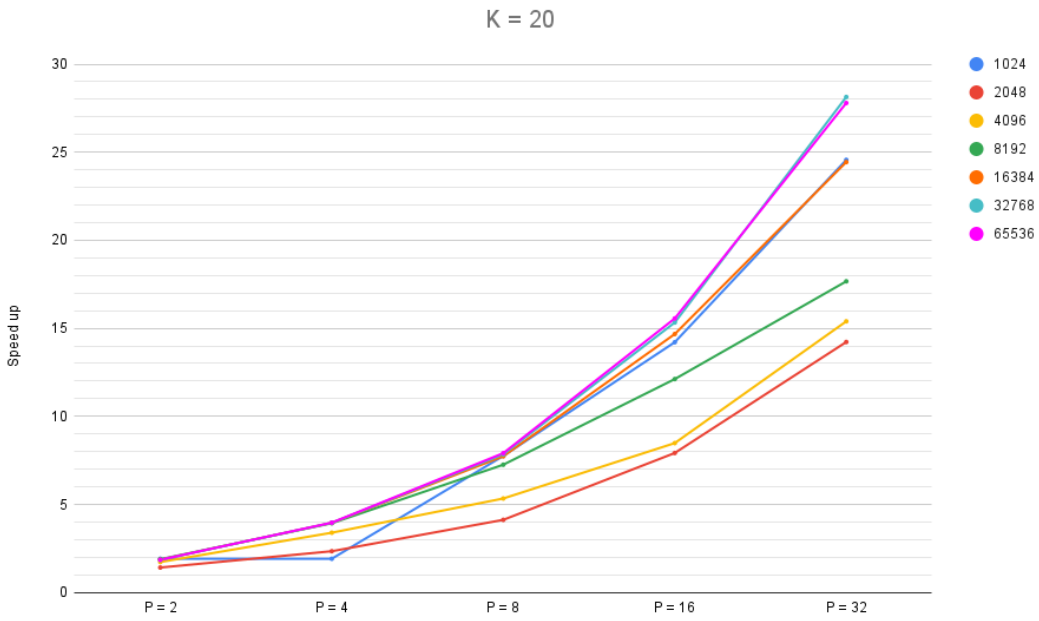

Figure 4.9: Speed-Up V2.0 with K = 20

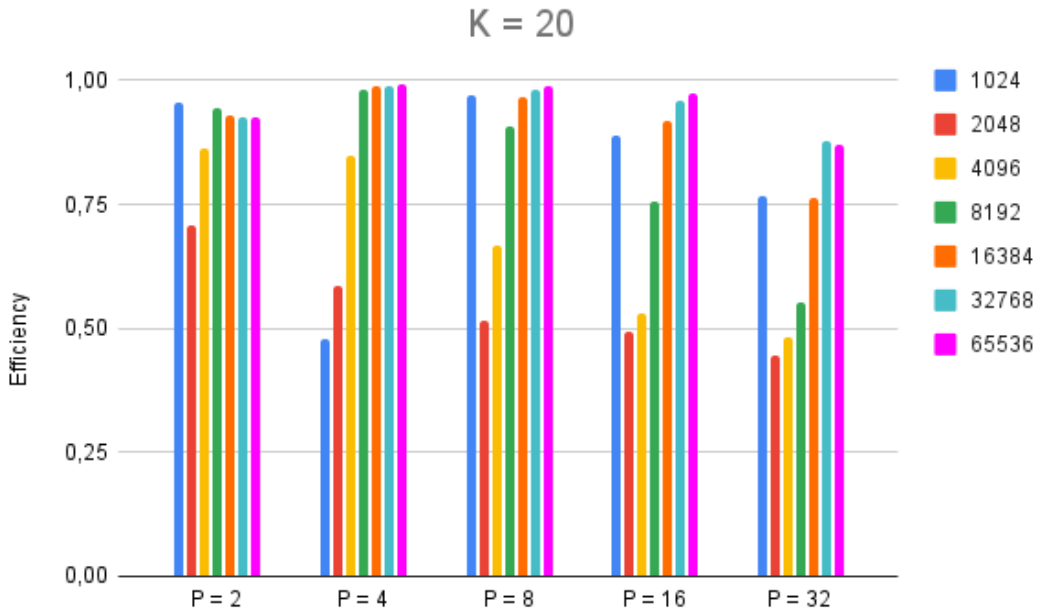

Figure 4.10: Efficiency V2.0 with K = 20

5. KNN V2.1

In this version, we have retained the same functionalities as in version 2.0 (Chapter 4). However, we have modified the sorting algorithm, opting for Merge Sort (Code 5.1).

Code 5.1: Merge Sort in V2

```
int find_position(double *array, int left, int right, double
    to_insert){
    while (left <= right){
        int mid = (left + right) / 2;

        if (array[mid] <= to_insert){
            left = mid + 1;
        }
        else{
            right = mid - 1;
        }
    }
    return left;
}
```

As demonstrated by result analysis, this algorithm performs better for large values of K (in our case, $K >= 10$).
In this chapter, we omit the description as previously stated. Nonetheless, the sections 4.1 and 4.2 remain relevant for this context.

5.1 Parallel V2.1.1

We've also implemented a new version of the aforementioned parallel algorithm, which focuses on adopting the communication and computation mechanisms we've already discussed. For the first version, it utilizes to the topologies described in subsection 3.3.1, while for the second version, it adopts the topologies outlined in subsection 3.3.2. As for the calculation part, it follows the algorithm described above (Chapter 5).
The communication differs only in the initial phase, specifically when the points are divided among the processes using sends and receives (see Code 3.6). In the updated version, a collective operation called *MPI Scatter* is utilized, which automatically distributes points from the Coordinator process to the others (see Code 5.2).

Code 5.2: Scatter in V2.1

```
if (my_rank == COORDINATOR)
    {
    //Coordinator sends the portion of points to each process
```

```
    MPI_Scatter(points, points_per_process, point_type,
        my_points, points_per_process, point_type, COORDINATOR,
        MPI_COMM_WORLD);
    MPI_Scatter(points, points_per_process, point_type,
        received_points, points_per_process, point_type,
        COORDINATOR, MPI_COMM_WORLD);
}
```

Indeed, this function eliminates the need to individually send and receive data for each process, enabling the same task to be accomplished in just one step.

5.2 Results and Performance evaluation

Just as in Chapter 3 (Section 3.4), we will demonstrate the Speedup and Efficiency of this version compared to the sequential one, using the same formulas as before. Below are the charts displaying all recorded values during the execution of both programs. Additionally, in Section 6.5, the execution time tables are provided.

5.2.1 Algorithm 2.1.0 Performance Charts

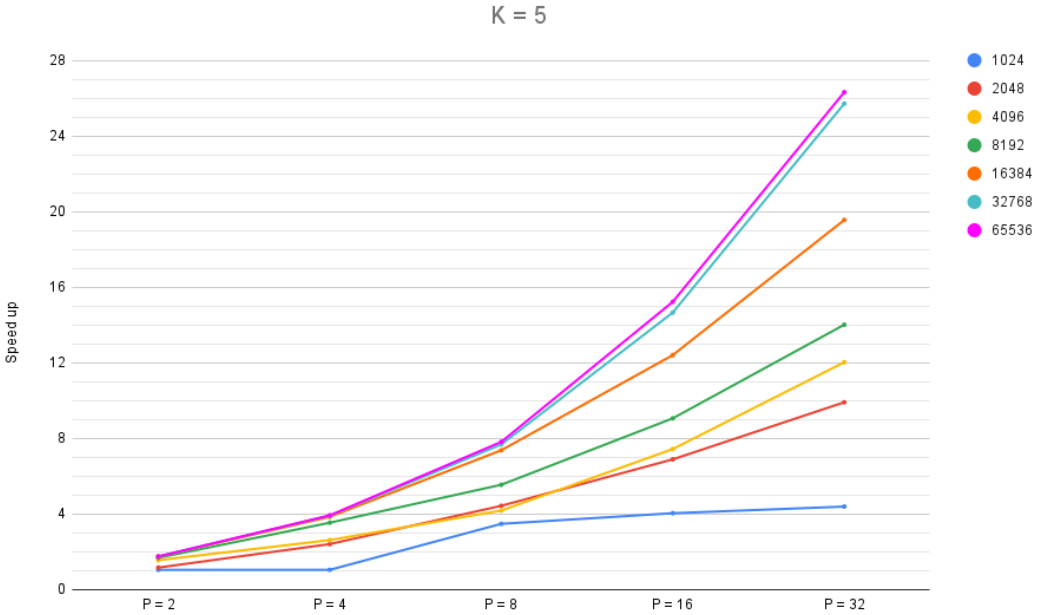

Figure 5.1: Speed-Up V2.1 with K = 5

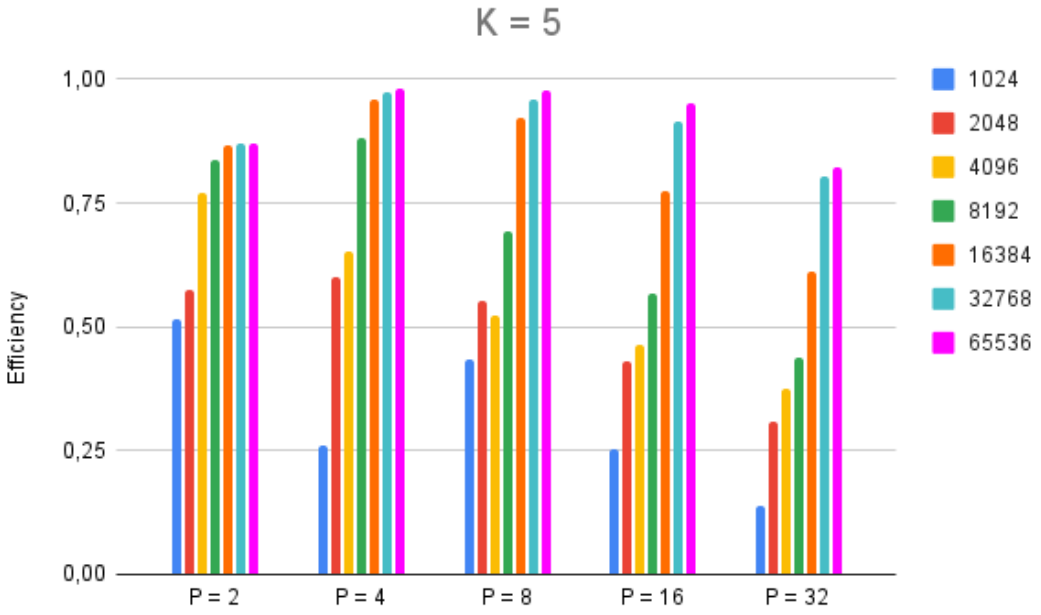

Figure 5.2: Efficiency V2.1 with K = 5

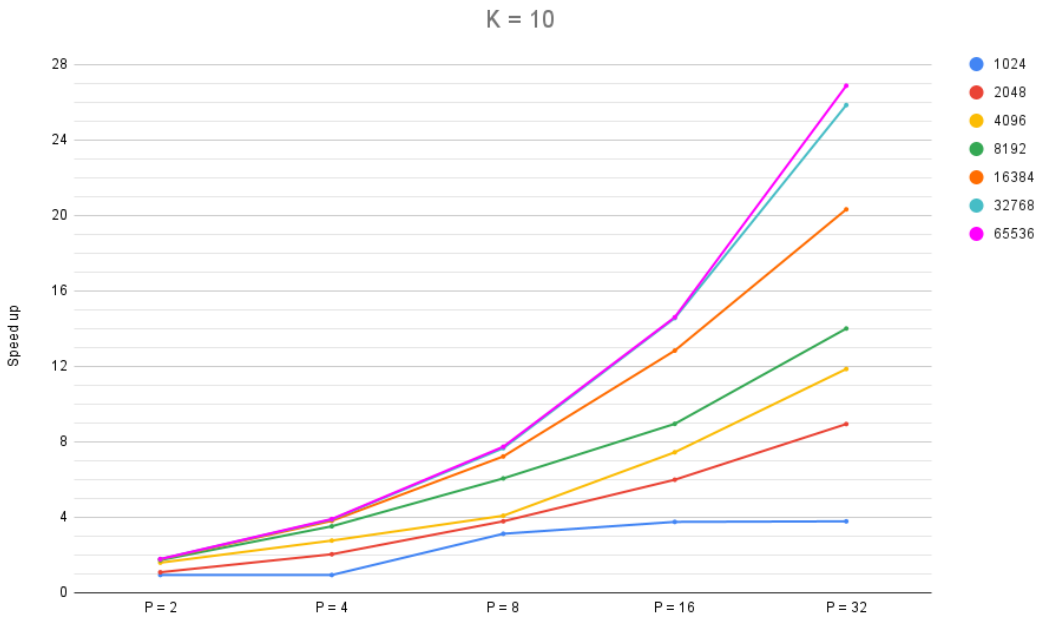

Figure 5.3: Speed-Up V2.1 with K = 10

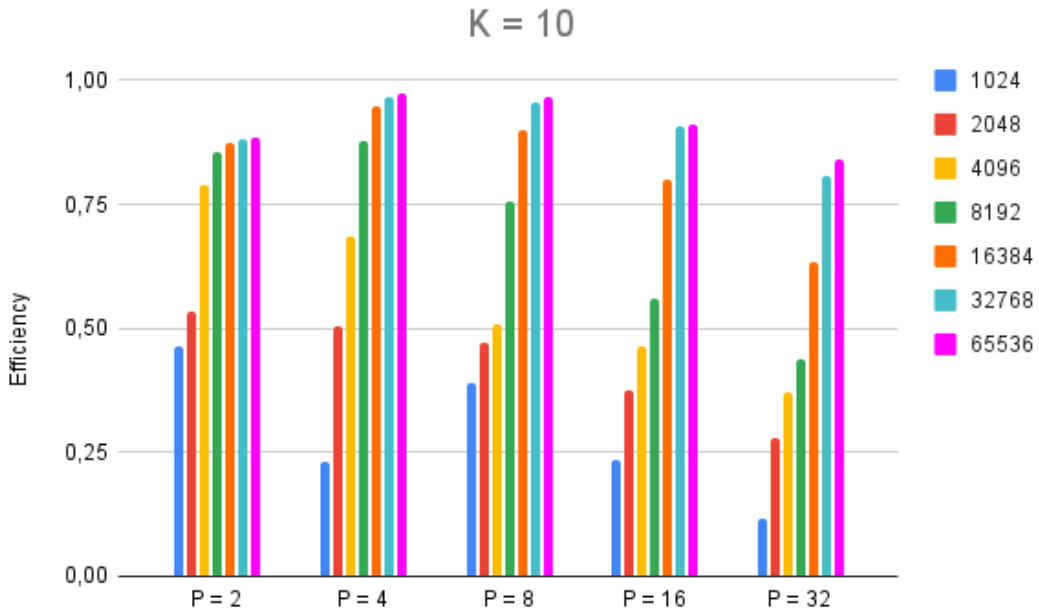

Figure 5.4: Efficiency V2.1 with K = 10

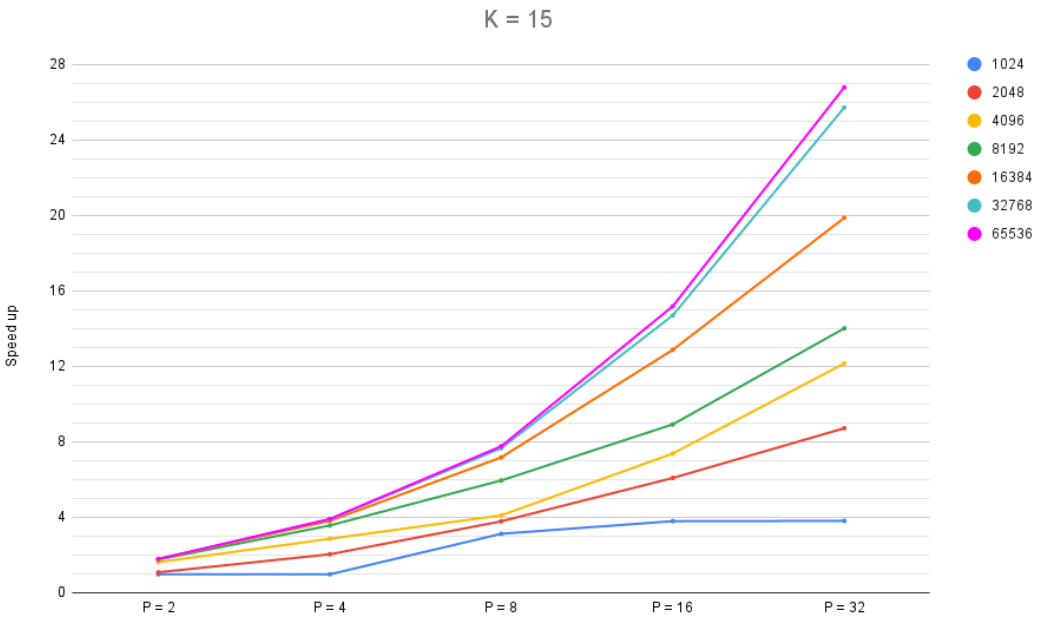

Figure 5.5: Speed-Up V2.1 with K = 15

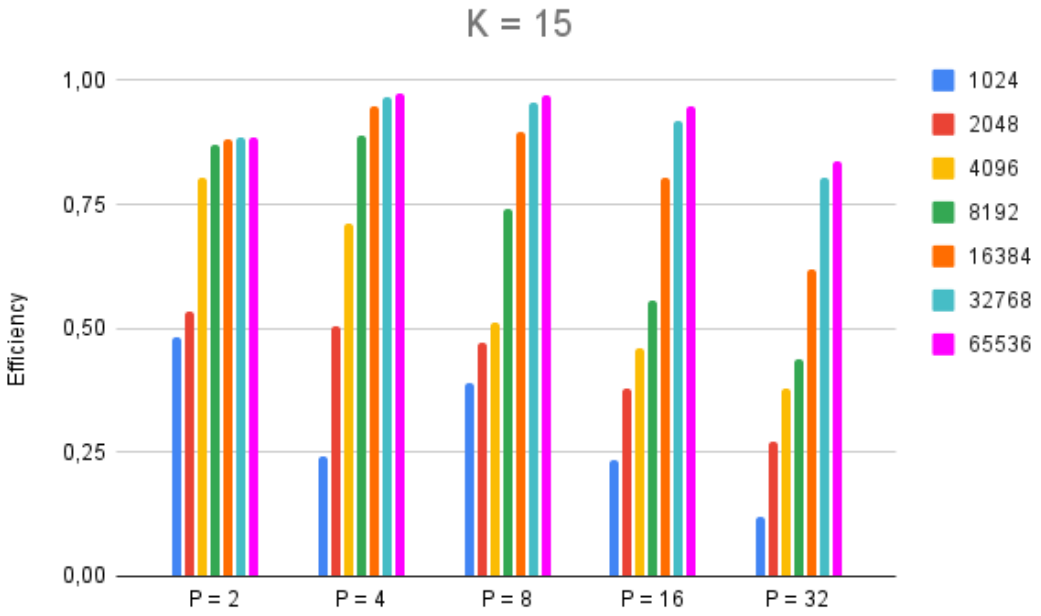

Figure 5.6: Efficiency V2.1 with K = 15

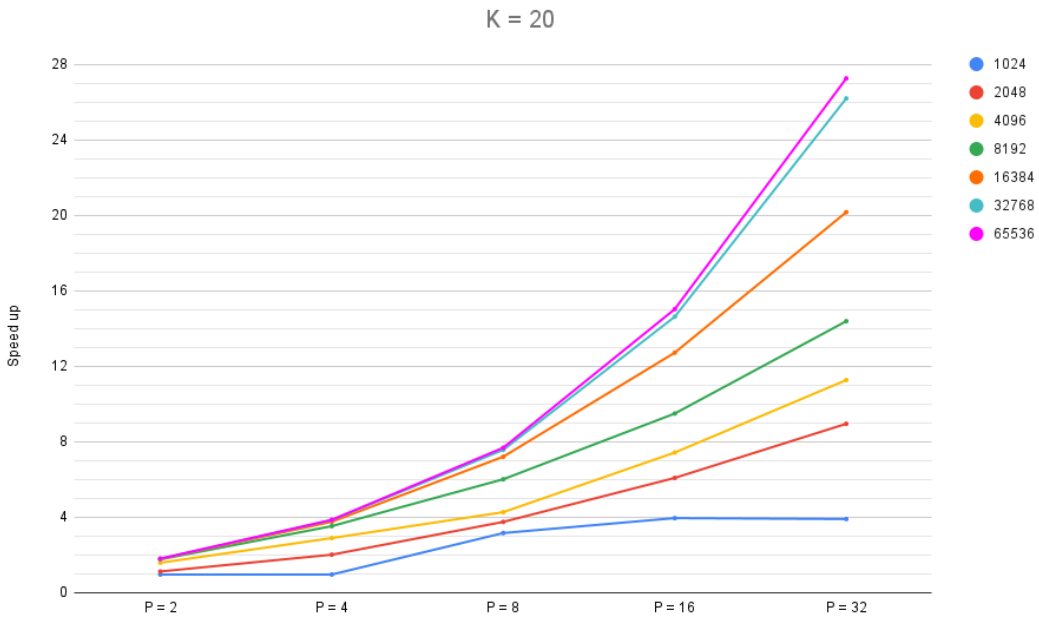

Figure 5.7: Speed-Up V2.1 with K = 20

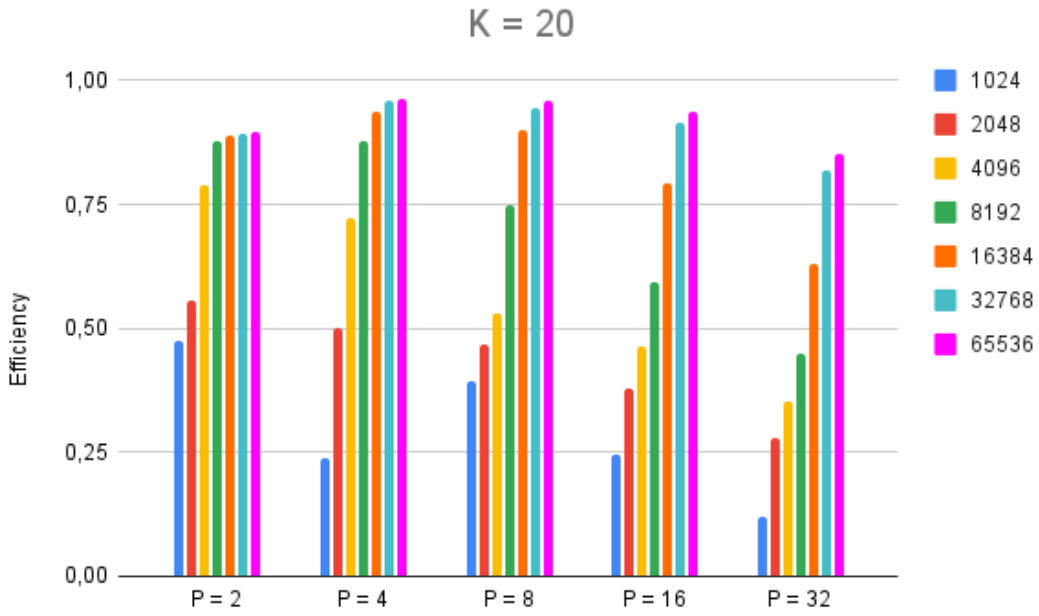

Figure 5.8: Efficiency V2.1 with K = 20

5.2.2 Algorithm 2.1.1 Performance Charts

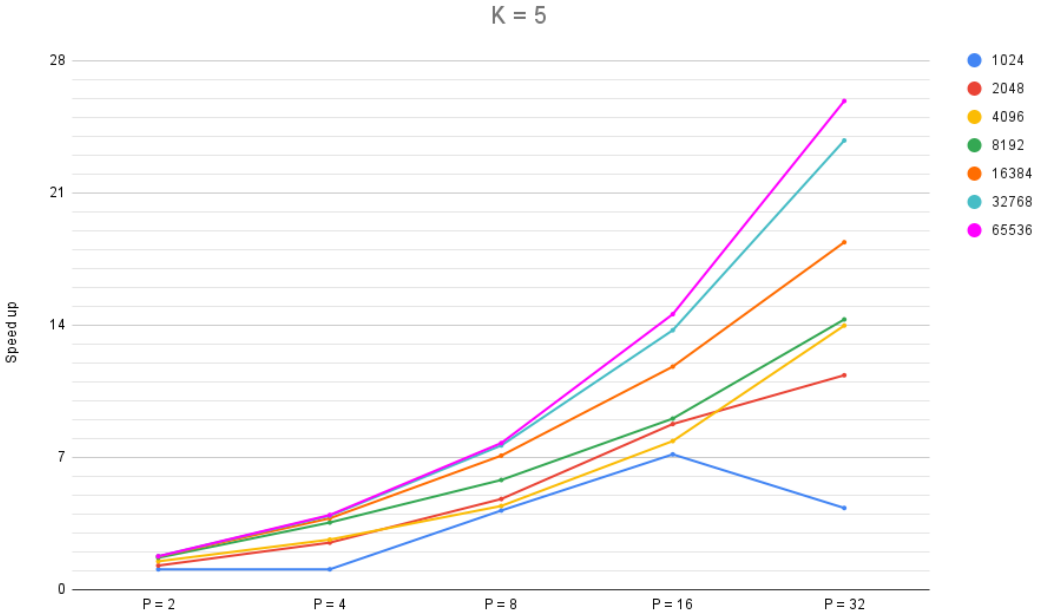

Figure 5.9: Speed-Up V2.1.1 with K = 5

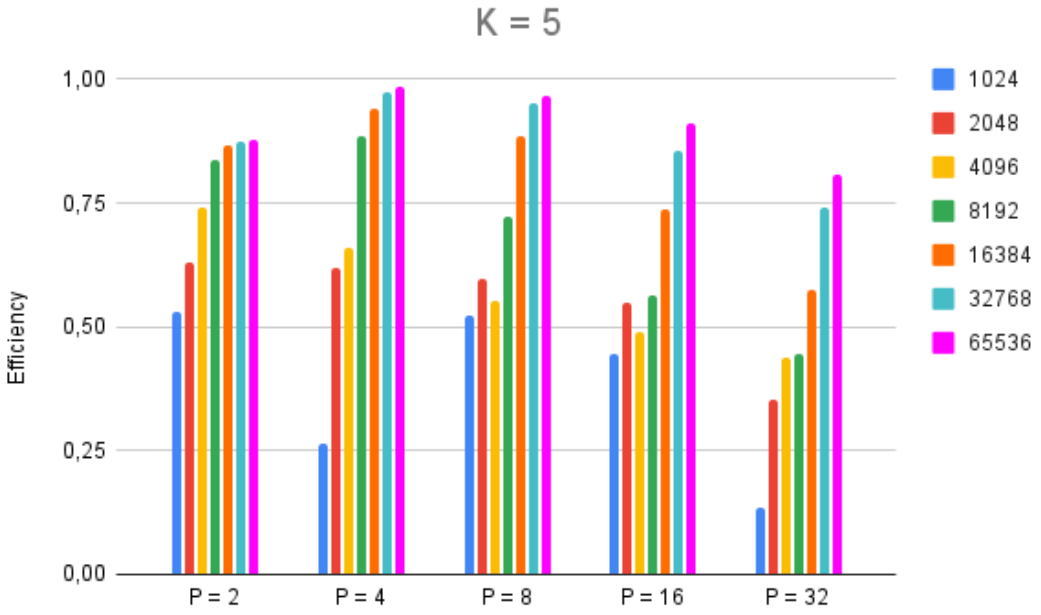

Figure 5.10: Efficiency V2.1.1 with K = 5

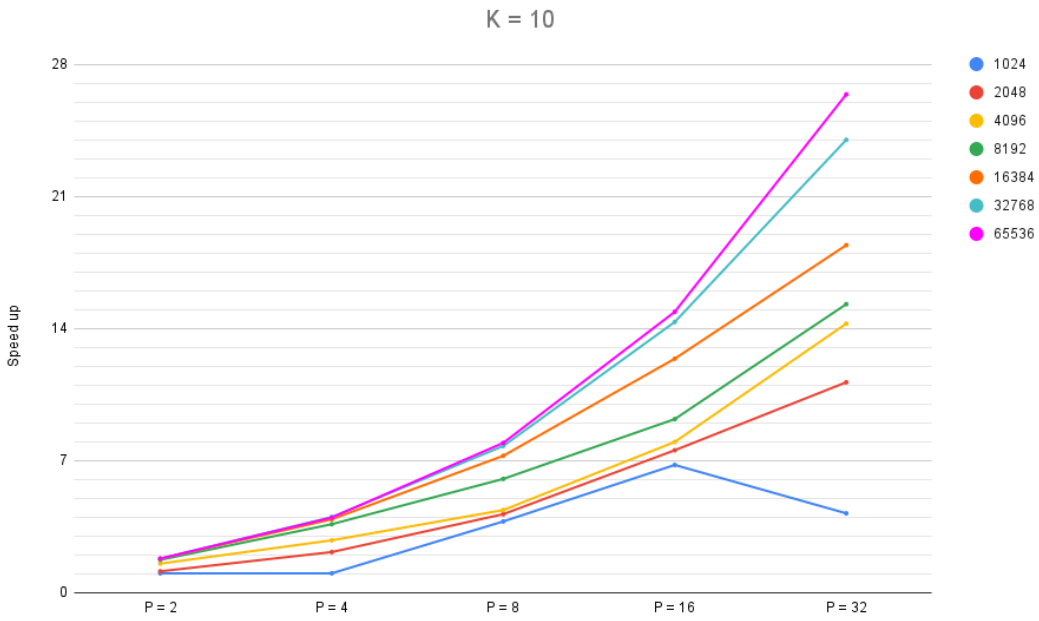

Figure 5.11: Speed-Up V2.1.1 with K = 10

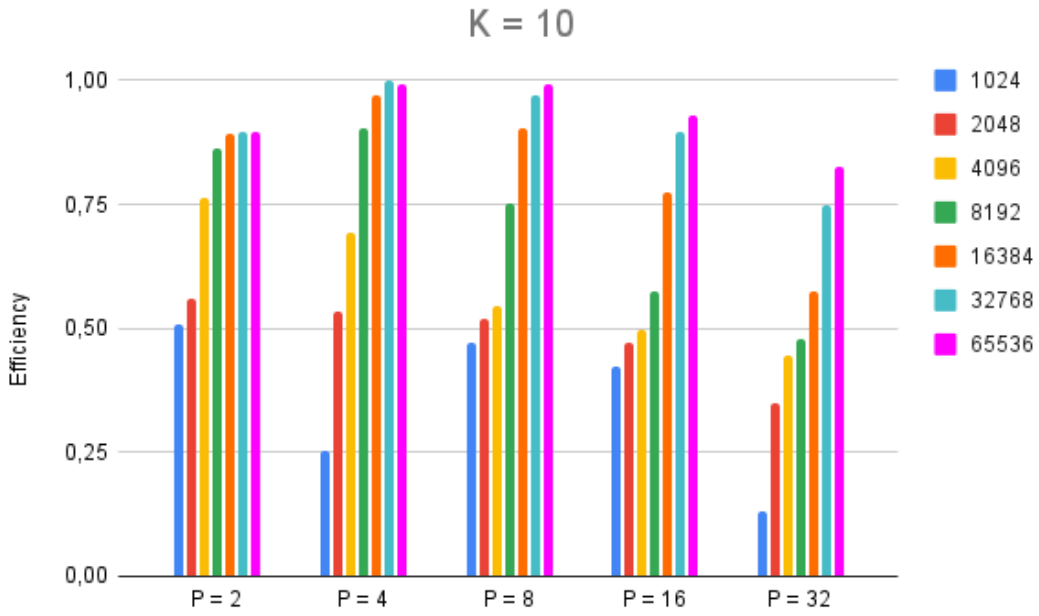

Figure 5.12: Efficiency V2.1.1 with K = 10

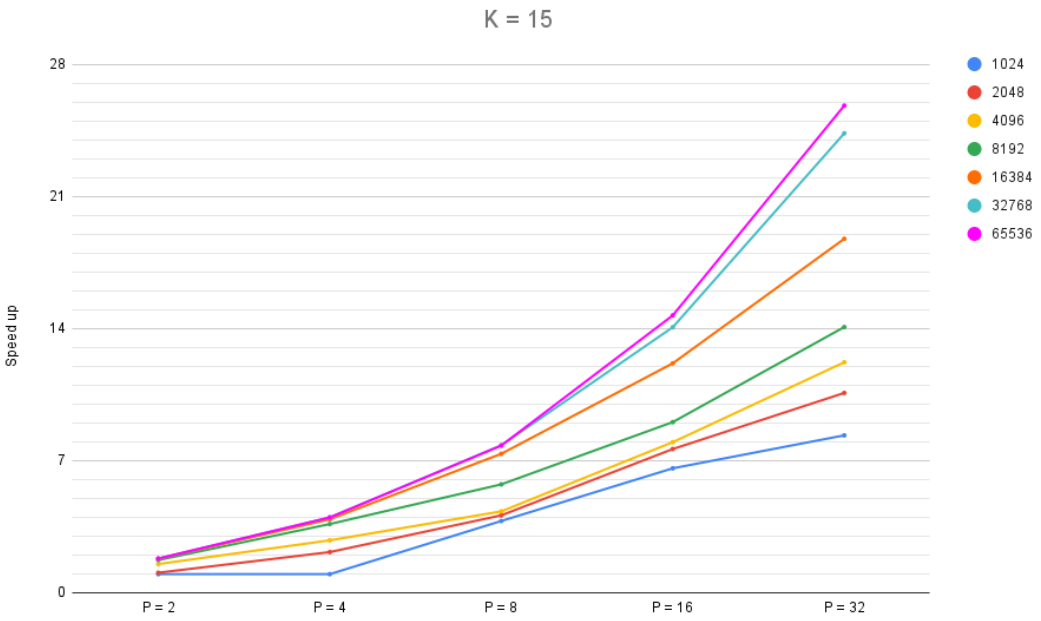

Figure 5.13: Speed-Up V2.1.1 with K = 15

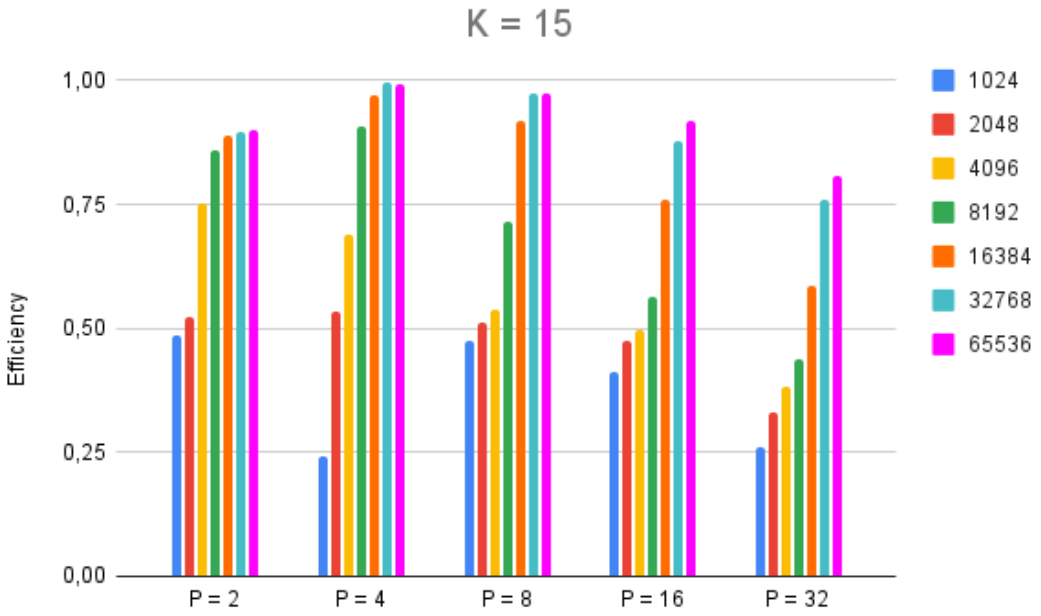

Figure 5.14: Efficiency V2.1.1 with K = 15

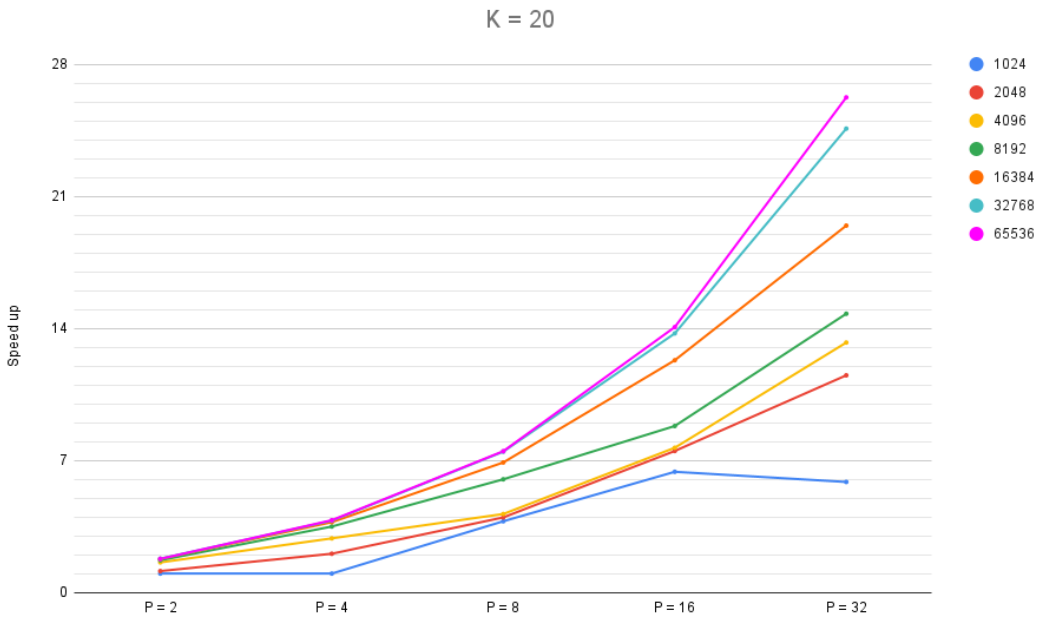

Figure 5.15: Speed-Up V2.1.1 with K = 20

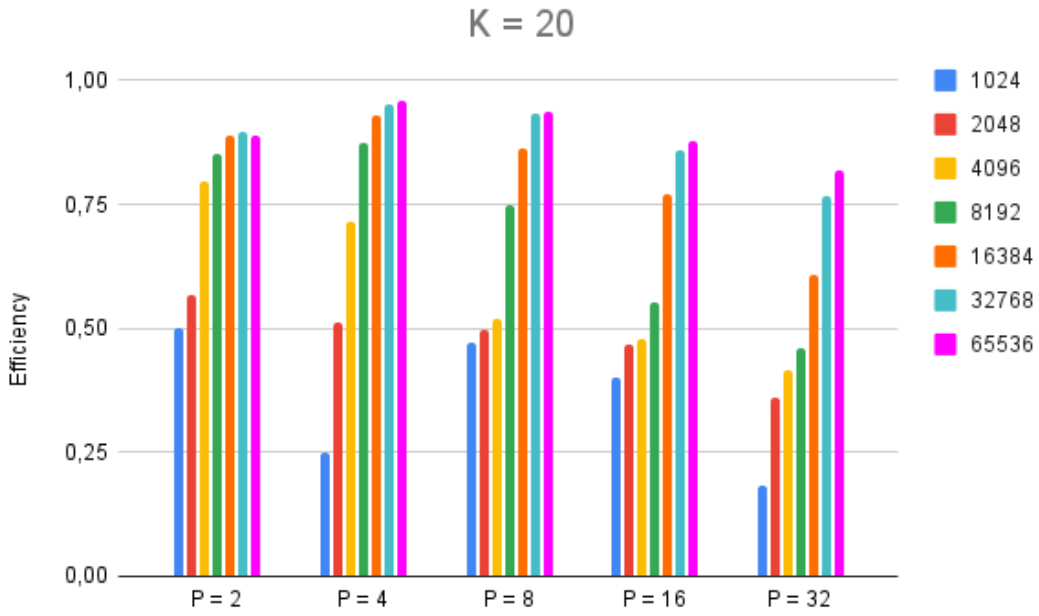

Figure 5.16: Efficiency V2.1.1 with K = 20

5.2.3 Algorithm 2.1.1 Ring Performance Charts

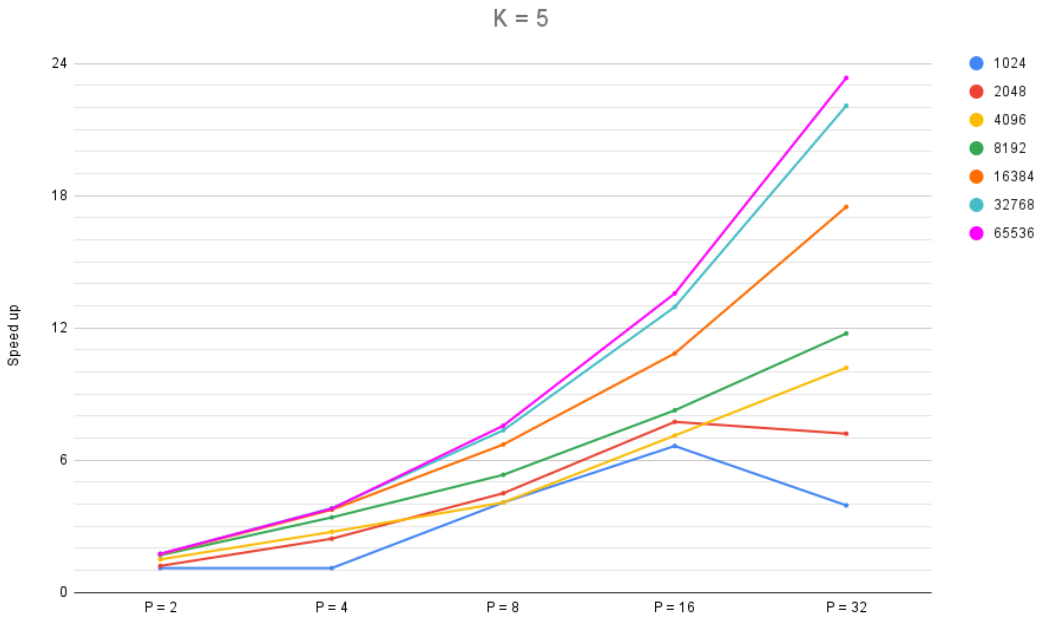

Figure 5.17: Speed-Up V2.1.1-ring with K = 5

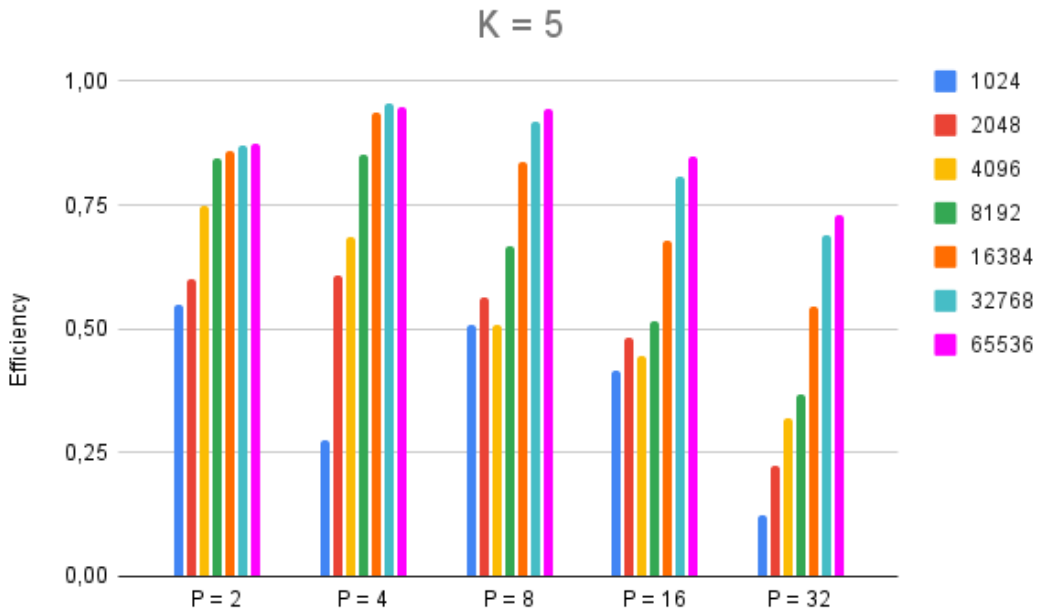

Figure 5.18: Efficiency V2.1.1-ring with K = 5

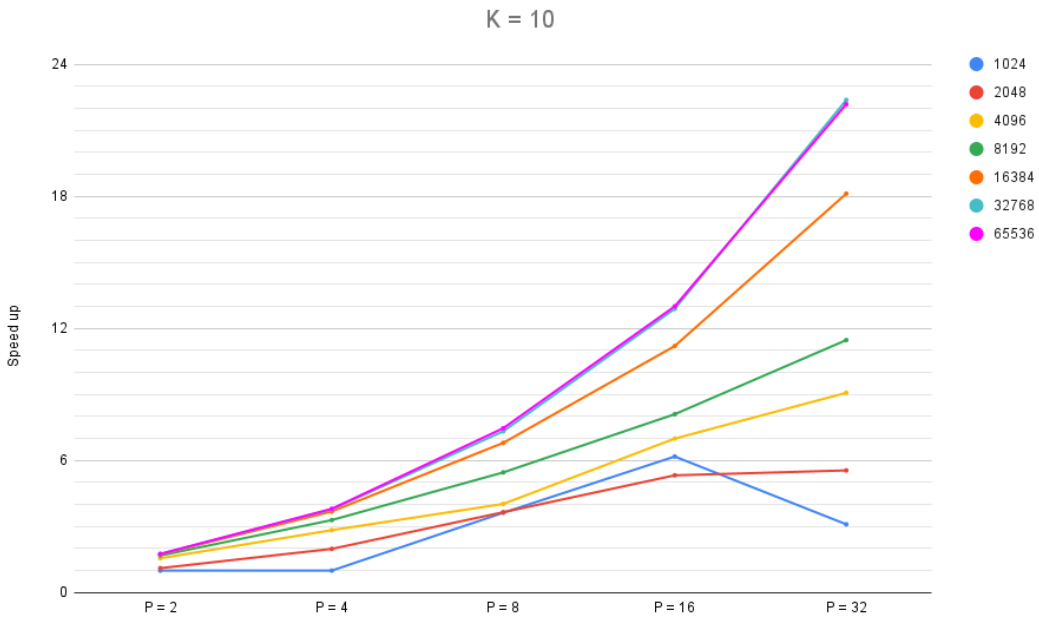

Figure 5.19: Speed-Up V2.1.1-ring with K = 10

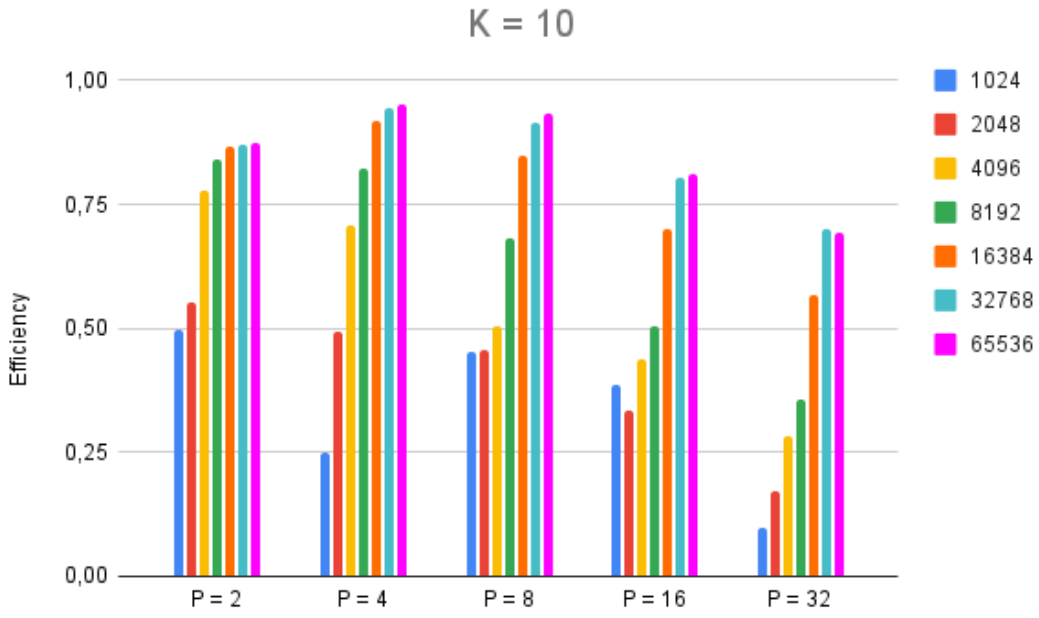

Figure 5.20: Efficiency V2.1.1-ring with K = 10

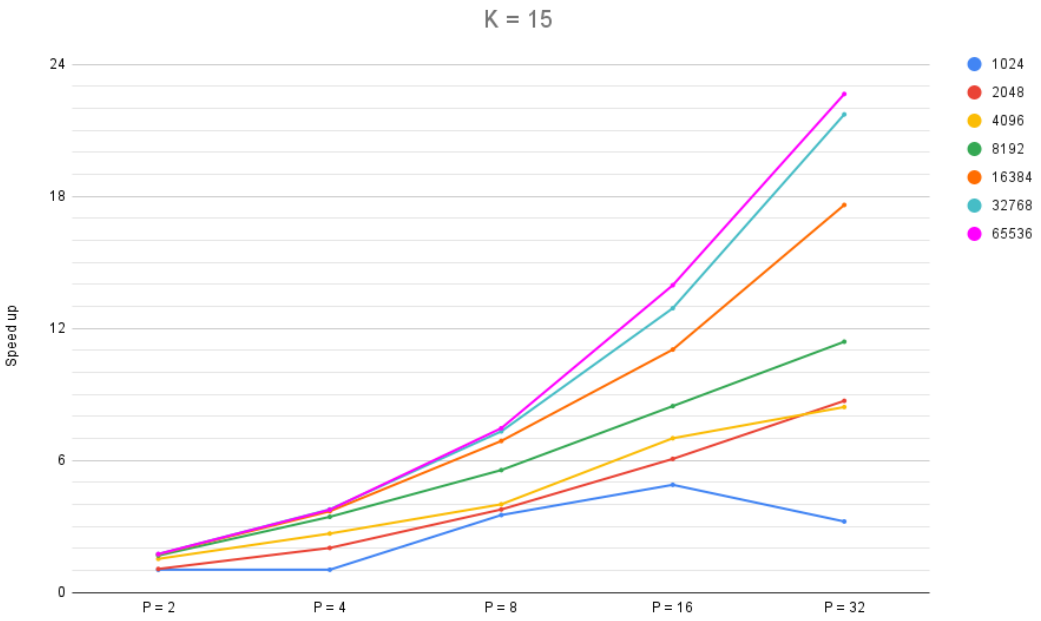

Figure 5.21: Speed-Up V2.1.1 with K = 15

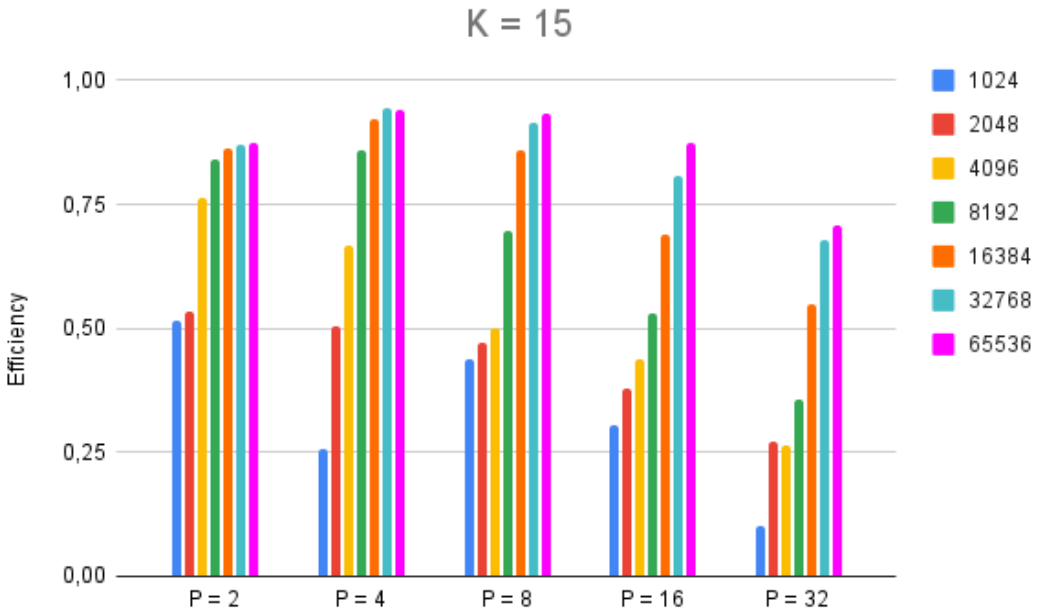

Figure 5.22: Efficiency V2.1.1-ring with K = 15

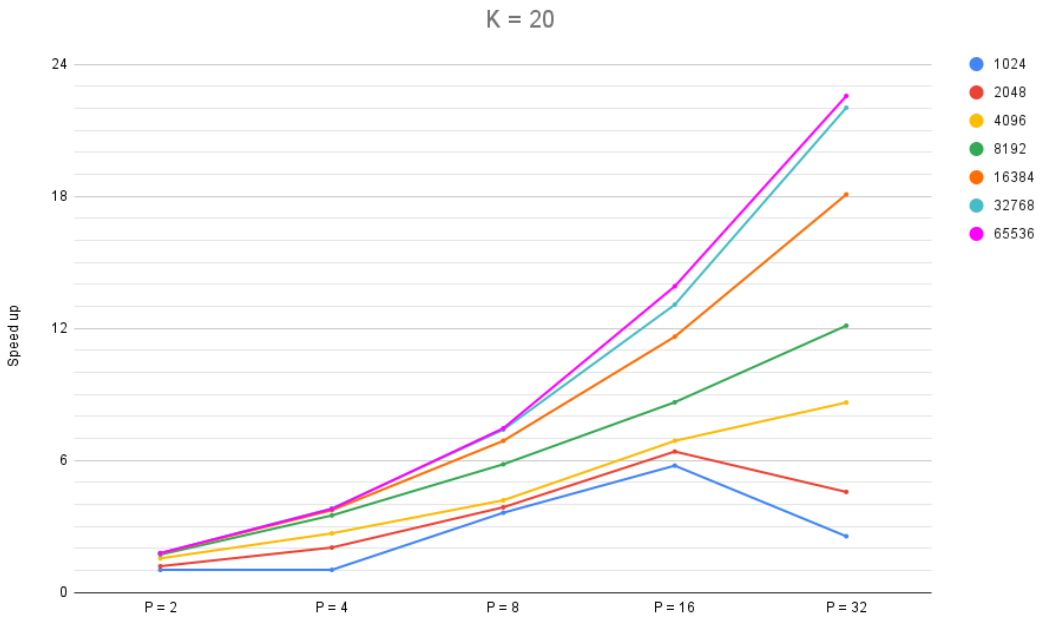

Figure 5.23: Speed-Up V2.1.1-ring with K = 20

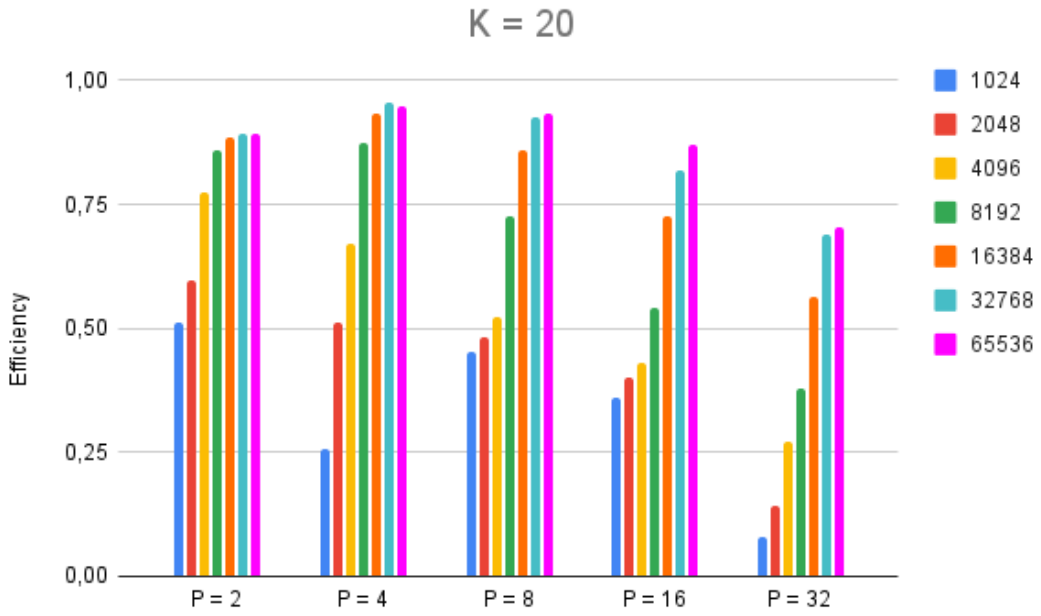

Figure 5.24: Efficiency V2.1.1-ring with K = 20

6. Execution Time Tables

6.1 Knn V1.0

N = 1024						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	0,144658	0,084777	0,036772	0,021303	0,011512	0,007809
10	0,096078	0,115761	0,050187	0,026969	0,01564	0,00982
15	0,158259	0,156307	0,068545	0,037276	0,02039	0,01243
20	0,200683	0,186049	0,082351	0,04347	0,024410	0,01455

N = 2048						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	0,344584	0,304361	0,139230	0,072005	0,039749	0,024106
10	0,372283	0,361607	0,188377	0,096888	0,05224	0,03084
15	0,501345	0,454717	0,258252	0,132364	0,07052	0,04127
20	0,632882	0,507309	0,273724	0,156062	0,08171	0,04924

N = 4096						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	1,140112	0,827517	0,411502	0,270289	0,139187	0,085241
10	1,459238	1,046274	0,527963	0,347851	0,18742	0,11197
15	1,953364	1,335189	0,684547	0,442483	0,2539	0,14896
20	2,311201	1,605240	0,756793	0,498308	0,3035	0,17617

N = 8192						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	4,476664	2,907417	1,320886	0,765192	0,510826	0,314782
10	5,867699	3,784006	1,681464	0,98504	0,6309	0,40323
15	7,751755	4,990688	2,277144	1,245636	0,75991	0,52248
20	9,09978	5,837823	2,638281	1,396991	0,85368	0,59789

N = 16384						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	17,787686	11,26098	4,744665	2,534153	1,36999	0,96989
10	23,022024	14,6339	6,29641	3,27436	1,8144	1,1571
15	30,738319	19,5399	8,54923	4,39584	2,4145	1,5111
20	36,062266	22,8982	10,0562	5,15653	2,7851	1,7233

N = 32768						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	70,826721	44,86563	18,439460	9,326640	4,91048	2,97154
10	91,932909	58,1761	24,586	12,3682	6,5027	3,8291
15	122,425802	76,9915	33,5922	16,9106	8,8933	5,1595
20	143,463893	90,6709	39,5517	19,8567	10,44	5,9879

N = 65536						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	284,267072	180,5312	73,409450	36,71897	18,9362	11,6167
10	368,718733	232,952	97,578	48,8922	25,22	14,877
15	489,958788	309,069	133,517	66,8671	34,69	19,996
20	575,050379	360,866	157,453	78,7612	40,912	23,595

6.2 Knn V1.1

N = 1024						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	0,144658	0,080824	0,034618	0,021449	0,013047	0,009877
10	0,096078	0,108315	0,047774	0,028522	0,01612	0,01262
15	0,158259	0,149662	0,06683	0,037512	0,02202	0,01398
20	0,200683	0,178795	0,081089	0,044653	0,026160	0,01644

N = 2048						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	0,344584	0,301889	0,134445	0,069208	0,040233	0,024893
10	0,372283	0,370048	0,183112	0,093187	0,05153	0,03152
15	0,501345	0,441404	0,254364	0,129647	0,06959	0,04173
20	0,632882	0,509742	0,288389	0,155444	0,08367	0,05057

N = 4096						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	1,140112	0,805677	0,403998	0,265427	0,138839	0,084816
10	1,459238	1,010501	0,493754	0,340495	0,18384	0,11205
15	1,953364	1,319481	0,692613	0,442271	0,25361	0,15215
20	2,311201	1,573079	0,736336	0,500652	0,30234	0,17978

N = 8192						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	4,476664	2,815694	1,277159	0,79858	0,530304	0,327829
10	5,867699	3,685501	1,644168	0,974303	0,60926	0,42273
15	7,751755	4,927165	2,241121	1,299688	0,81348	0,55828
20	9,09978	5,806074	2,609398	1,487472	0,94688	0,63735

N = 16384						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	17,787686	10,9445	4,620772	2,463698	1,45673	1,11495
10	23,022024	14,3132	6,11312	3,25963	1,893	1,3449
15	30,738319	19,1227	8,44816	4,37999	2,5082	1,7238
20	36,062266	22,5575	10,00074	5,20801	2,883	1,9288

N = 32768						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	70,826721	43,66499	18,037070	9,201514	4,975	3,25709
10	91,932909	57,197	24,035	12,1948	6,6467	4,1665
15	122,425802	76,0219	33,2081	16,7849	8,9881	5,4097
20	143,463893	89,2909	39,3604	19,9115	10,531	6,2438

N = 65536						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	284,267072	174,9916	71,791490	35,92027	18,7452	11,4798
10	368,718733	228,884	95,7661	47,946	25,115	15,011
15	489,958788	303,801	132,789	66,2907	34,256	20,095
20	575,050379	356,285	156,492	78,4689	40,524	23,533

6.3 Knn V1.1 ring

N = 1024						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	0,144658	0,074017	0,034786	0,018709	0,011637	0,021503
10	0,096078	0,104235	0,047963	0,025703	0,015933	0,020183
15	0,158259	0,150345	0,066952	0,035892	0,021067	0,029283
20	0,200683	0,176762	0,080477	0,042253	0,025467	0,0243

N = 2048						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	0,344584	0,31285	0,137935	0,069818	0,045597	0,034775
10	0,372283	0,368267	0,18435	0,097052	0,056967	0,045133
15	0,501345	0,439735	0,249523	0,128953	0,075833	0,06135
20	0,632882	0,496722	0,266402	0,166532	0,088767	0,083867

N = 4096						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	1,140112	0,83301	0,435963	0,276007	0,158302	0,098963
10	1,459238	1,054492	0,529893	0,368508	0,201217	0,154417
15	1,953364	1,35176	0,69825	0,485903	0,2766	0,1744
20	2,311201	1,585700	0,8846	0,54747	0,33575	0,209467

N = 8192						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	4,476664	2,940395	1,330671	0,868378	0,557752	0,378397
10	5,867699	3,783852	1,701785	1,004382	0,714467	0,435733
15	7,751755	5,038468	2,314688	1,342255	0,8698	0,5739
20	9,09978	5,930495	2,735172	1,581522	1,03295	0,610867

N = 16384						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	17,787686	11,3708	4,772790	2,653342	1,705433	0,965417
10	23,022024	14,765833	6,324183	3,432983	2,1055	1,241
15	30,738319	19,576667	8,549867	4,53695	2,730333	1,5715
20	36,062266	23,141833	10,281833	5,446633	3,192667	1,925167

N = 32768						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	70,826721	45,083633	18,423600	9,577895	5,741617	3,609983
10	91,932909	58,785833	24,3995	12,633	7,382167	4,863333
15	122,425802	78,139333	33,4365	18,636333	10,121333	5,846167
20	143,463893	91,698333	39,695667	20,454167	11,725	6,904667

N = 65536						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	284,267072	183,720667	72,719250	38,962317	22,524	12,725833
10	368,718733	235,623333	97,223833	49,899167	29,01	16,908333
15	489,958788	313,076667	133,455	67,59	39,361667	22,375
20	575,050379	366,446667	156,681667	79,240833	44,033333	25,658333

6.4 Knn V2.0

N = 1024						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	0,158435	0,082889	0,04004	0,021456	0,01277	0,008243
10	0,207263	0,121714	0,056899	0,030775	0,01722	0,01058
15	0,164345	0,172324	0,081771	0,042982	0,02395	0,01391
20	0,407975	0,213115	0,101315	0,052635	0,02871	0,01661

N = 2048						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	0,436831	0,293532	0,150709	0,077875	0,042211	0,023971
10	0,63304	0,374521	0,215507	0,110601	0,05882	0,03432
15	0,644736	0,486031	0,286667	0,158585	0,08258	0,04728
20	0,800978	0,565908	0,342079	0,19425	0,10114	0,05632

N = 4096						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	1,357343	0,830567	0,440118	0,291295	0,153711	0,085749
10	2,028402	1,084432	0,593965	0,385424	0,21629	0,12238
15	2,559685	1,515353	0,834732	0,515944	0,30531	0,17212
20	3,204716	1,855288	0,944576	0,600611	0,37786	0,20817

N = 8192						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	4,910603	2,806902	1,414704	0,861972	0,516046	0,31937
10	7,658495	4,043033	1,921733	1,092774	0,67516	0,44159
15	10,147103	5,68099	2,759927	1,498156	0,88775	0,60889
20	13,02239	6,899668	3,313124	1,795182	1,07475	0,73712

N = 16384						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	19,053536	10,98609	5,221644	2,738802	1,51909	1,01925
10	29,056923	15,9852	7,2643	3,77842	2,0608	1,311
15	40,452834	22,2864	10,4807	5,37431	2,8987	1,7323
20	50,287056	27,0287	12,7198	6,50318	3,4271	2,0578

N = 32768						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	75,893376	43,82734	20,330850	10,31149	5,36103	3,00526
10	112,608757	63,9414	28,5367	14,4015	7,426	4,2162
15	161,229141	88,4295	41,282	20,777	10,663	5,8753
20	198,804962	107,318	50,2643	25,3491	12,97	7,0661

N = 65536						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	305,027656	175,3556	80,766620	40,53043	20,6211	11,3651
10	449,431124	255,792	113,557	56,9464	28,99	16,374
15	645,743103	352,905	164,508	82,4416	41,865	23,222
20	793,685751	427,956	200,232	100,33	51,008	28,557

6.5 Knn V2.1

N = 1024						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	0,089918	0,086927	0,04251	0,025925	0,022307	0,020522
10	0,090288	0,097577	0,048079	0,029023	0,02414	0,02395
15	0,093502	0,097233	0,050125	0,030069	0,02474	0,02465
20	0,105134	0,110604	0,056134	0,033383	0,02668	0,02702

N = 2048						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	0,372504	0,323225	0,155405	0,08423	0,054141	0,037624
10	0,35523	0,331683	0,175474	0,094161	0,05946	0,0398
15	0,363218	0,340137	0,179517	0,096337	0,05987	0,04173
20	0,401286	0,360967	0,200341	0,107273	0,0661	0,04489

N = 4096						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	1,271407	0,824276	0,487227	0,304794	0,171201	0,105766
10	1,407893	0,892016	0,512158	0,346241	0,18949	0,11888
15	1,423622	0,887062	0,500532	0,348192	0,19336	0,11729
20	1,566633	0,994832	0,543492	0,36819	0,21129	0,13917

N = 8192						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	5,07769	3,039427	1,438999	0,917147	0,561031	0,362566
10	5,599187	3,271104	1,596369	0,926248	0,62665	0,4003
15	5,645876	3,239084	1,59029	0,951164	0,6339	0,40334
20	6,176443	3,524062	1,756775	1,029347	0,65137	0,42962

N = 16384						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	20,303929	11,70388	5,291936	2,757661	1,63773	1,03855
10	22,340636	12,7638	5,88536	3,09892	1,7431	1,1004
15	22,415563	12,6951	5,91335	3,13068	1,7433	1,129
20	24,548723	13,8331	6,55162	3,41219	1,9317	1,2181

N = 32768						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	81,180319	46,56704	20,825340	10,58294	5,54161	3,15824
10	89,26141	50,5757	23,0892	11,6726	6,1388	3,4557
15	89,492086	50,6313	23,1139	11,7006	6,0942	3,482
20	97,738463	54,7846	25,5049	12,9339	6,6881	3,7332

N = 65536						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	324,885227	186,5848	82,905070	41,61797	21,3486	12,3482
10	356,715942	201,298	91,6459	46,1839	24,455	13,283
15	357,577663	201,975	91,7748	46,1314	23,567	13,357
20	390,247904	218,004	101,343	50,8765	25,989	14,325

6.6 Knn V2.1.1

N = 1024						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	0,089918	0,084721	0,03857	0,021545	0,012584	0,020877
10	0,090288	0,088942	0,042969	0,023986	0,01336	0,02153
15	0,093502	0,096396	0,044913	0,024683	0,01421	0,01123
20	0,105134	0,105066	0,050782	0,027862	0,01644	0,01795

N = 2048						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	0,372504	0,295638	0,150623	0,077878	0,042569	0,032872
10	0,35523	0,316649	0,165869	0,085644	0,04713	0,03189
15	0,363218	0,34703	0,169703	0,088882	0,04779	0,03435
20	0,401286	0,353943	0,195465	0,100912	0,05352	0,03487

N = 4096						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	1,271407	0,85643	0,482663	0,288045	0,161973	0,091066
10	1,407893	0,921902	0,509277	0,322406	0,17651	0,09882
15	1,423622	0,946021	0,515225	0,331271	0,17871	0,1167
20	1,566633	0,983502	0,546504	0,3768	0,2043	0,11828

N = 8192						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	5,07769	3,033991	1,434885	0,877031	0,561879	0,355456
10	5,599187	3,240348	1,547257	0,930354	0,60944	0,36646
15	5,645876	3,289287	1,557132	0,985639	0,62538	0,40141
20	6,176443	3,621902	1,767778	1,029846	0,70018	0,4181

N = 16384						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	20,303929	11,69016	5,399048	2,86895	1,72239	1,1053
10	22,340636	12,5145	5,76004	3,08519	1,8033	1,2137
15	22,415563	12,6151	5,77925	3,05046	1,8458	1,196
20	24,548723	13,8323	6,59167	3,56315	1,9948	1,2627

N = 32768						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	81,180319	46,38683	20,837120	10,65919	5,91888	3,41729
10	89,26141	49,8152	22,325	11,4998	6,2268	3,7211
15	89,492086	49,8949	22,5016	11,4747	6,3674	3,6781
20	97,738463	54,6311	25,6997	13,0996	7,1191	3,9759

N = 65536						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	324,885227	185,3773	82,576100	41,95962	22,3089	12,5707
10	356,715942	198,903	89,8287	45,0035	23,985	13,514
15	357,577663	199,04	89,9231	45,9372	24,354	13,86
20	390,247904	219,184	101,9237	52,1281	27,749	14,874

6.7 Knn V2.1.1 ring

N = 1024						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	0,089918	0,081709	0,03910	0,022052	0,013542	0,022782
10	0,090288	0,090921	0,045395	0,024939	0,01462	0,02917
15	0,093502	0,090616	0,047376	0,026589	0,01912	0,029
20	0,105134	0,102505	0,052329	0,029032	0,01825	0,04119

N = 2048						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	0,372504	0,310525	0,153036	0,082826	0,04816	0,051744
10	0,35523	0,321672	0,179485	0,097422	0,06676	0,06407
15	0,363218	0,340137	0,179517	0,096337	0,05987	0,04173
20	0,401286	0,336951	0,19667	0,103706	0,06269	0,08787

N = 4096						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	1,271407	0,850613	0,463312	0,312011	0,178741	0,124813
10	1,407893	0,906896	0,497486	0,349963	0,20141	0,15523
15	1,423622	0,932566	0,532106	0,355509	0,2032	0,16903
20	1,566633	1,011832	0,583578	0,374163	0,22751	0,18154

N = 8192						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	5,07769	3,011182	1,493061	0,952248	0,614898	0,432418
10	5,599187	3,333301	1,702706	1,026695	0,69144	0,48838
15	5,645876	3,363647	1,643155	1,015604	0,66696	0,49566
20	6,176443	3,592022	1,765048	1,060807	0,71486	0,50966

N = 16384						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	20,303929	11,83001	5,417178	3,027041	1,87328	1,16121
10	22,340636	12,905	6,07114	3,287	1,9955	1,2332
15	22,415563	12,976	6,07793	3,25716	2,0317	1,2734
20	24,548723	13,8584	6,56787	3,56487	2,1129	1,3577

N = 32768						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	81,180319	46,61396	21,242260	11,04083	6,27084	3,67787
10	89,26141	51,2442	23,6619	12,1926	6,9281	3,9905
15	89,492086	51,4049	23,7142	12,2316	6,9313	4,121
20	97,738463	54,8005	25,6239	13,2184	7,4757	4,4388

N = 65536						
K	Sequential	P = 2	P = 4	P = 8	P = 16	P = 32
5	324,885227	185,6214	85,574940	42,97297	23,9588	13,9236
10	356,715942	203,713	93,87	47,8092	27,446	16,084
15	357,577663	204,607	95,1613	47,9265	25,619	15,792
20	390,247904	218,317	102,903	52,3359	28,061	17,303

7. Conclusion

In conclusion, as demonstrated by the results, the 2.0 algorithm emerges as the top parallel algorithm due to its superior speedup and efficiency. However, when assessing based on execution time, the 2.1 algorithm outperforms others, establishing itself as the prime choice, primarily because it doesn't have to engage in communication between processors.
Given the superior speed of the 2.1 algorithm relative to others, it stands to reason that if compared against any of the others sequential one, it would likely exhibit greater speedup and efficiency than the other parallel ones. Conversely, if the other algorithms were compared to this sequential approach, their parallel implementations would likely show less efficiency and speedup.
At the end, we want to include a small note regarding the reliability of the execution results of the programs. Since the programs were executed on a shared multi-user machine, it's important to acknowledge that the evaluations may not be entirely accurate due to the potential for other users utilizing the shared resources.