Chapter 7

CUDA Accelerated Parallel Non-Dominated Sorting

7.1 Introduction

Advances in software and hardware introduce new ways of solving real-life industrial problems. Computational and hardware resources are more powerful than ever. This has allowed engineers and researchers to explore new problem-solving domains by incorporating methods that were previously considered 'too complex.'

With an increase in computational capability, researchers are able to tackle problems with large datasets efficiently. Multi-objective Optimisation Problems (MOOPs) are a part of this category. The solutions to these problems are represented as an array of values, one for each objective. Each value of the array depicts how well that solution satisfies the corresponding objective. One solution may be excellent at satisfying some objectives but worse at satisfying others. Non-dominated Sorting (NDS) is a pivotal step in Multi-objective Evolutionary Algorithms (MOEAs). An NDS algorithm sorts the solutions generated by MOEAs and ranks them based on their viability. NDS tends to be the most computationally heavy, time-consuming step of the algorithm, and as such, many researchers have attempted to speed it up in the past, such as [291, 292].

For most industry-scale problems, the number of solutions simultaneously being processed can be large-scale. Unfortunately, NDS algorithms tend to have quadratic time complexity in the worst case. Thus, reducing the complexity of NDS has been a key topic of research and a major challenge among the research community dealing with optimization problems using MOEAs. Since we cannot decide on a single most viable solution, NDS algorithms instead look for a particular group (or 'front') of solutions called the Pareto-optimal front of the set, such that:

- We cannot definitively say that one solution of the Pareto-optimal front is better than the another.
- There is always a solution on this front that is better than any other solution not belonging to this front.

Previous research has aimed at making the NDS algorithms more efficient by reducing the number of objective comparisons and removing redundancy, such as [291] etc. Other notable works include Distributed NSGA-II [293] and BOS [294]. They reduce the overall complexity from a naive cubic relation to a quadratic relation w.r.t the number of solutions. There has been a lack of research in exploring the scope of parallelism in these algorithms. Modern GPUs can run thousands of threads concurrently. Nowadays, they are extensively used for scientific computing.

7.2 Motivation and Contribution

Over the past few years, GPU usage for large-scale AI applications has become common. Many high-performance computing and cloud computing problems also use GPUs for general computing applications. NVIDIA's Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model that enables easy abstraction to GPU Hardware and general-purpose parallel computing on Nvidia GPUs. CUDA C++ is an extension of C++ that allows users to write code for parallel implementation of CUDA functions called kernels. Traditionally, most Pareto-based MOEAs

dissuade parallelism; thus, the challenge of reducing the overall computation time faced remains predominantly unmet. A few works have hypothesized parallel implementations, but most of them are theoretical. Others lack a sufficient amount of parallelism that can offset the extra time taken by CUDA to launch and synchronize kernels.

The major contributions of this chapter are as follows-

1. We analyze Corner Sort [295], an NDS algorithm, and highlight two areas within it with a high scope of parallelism.

2. We propose Parallel Corner Sort for efficiently parallelizing NDS.

3. Parallel Corner Sort parallelizes the Pareto-optimal solution finding as well as the dominance calculations in order to improve computation time. We accomplish this via CUDA.

4. A theoretical analysis of the complexity of Parallel Corner Sort is done for some general cases. We also explore the challenges faced by it.

5. We implemented the serial and parallel approaches in CUDA C++ and compared their performances on large population size data created artificially.

6. Experimental observations showed motivating results in terms of computation time with an increasing number of solutions. We observed that the time taken by Parallel Corner Sort does not vary much with the number of objectives.

7. We also observed that the effect of increasing the number of solutions does not affect the time taken by Parallel Corner Sort to the same extent as other algorithms.

7.3 Theoretical Background

7.3.1 Non-dominated Sorting

NDS is a type of high-dimensional sorting and is an essential part of **MOEAs**. Unlike single-objective optimization, the solution to a MOOP is a set of points called the **Pareto-Optimal Front**, where each solution is better than the other solutions in some way but worse in some other way. This optimal set of solutions is obtained by NDS, which is based on **Pareto dominance**.

For this chapter, we will denote the size of the multi-objective Population (solution space) with N and the number of objectives for each solution with m, unless specified otherwise. NDS is illustrated in Figure 7.1.

A **Front** is defined as a set of solutions that are non-dominated w.r.t each other. NDS algorithms divide the population into a set of indexed fronts. Each solution is assigned to a front such that all solutions in that front are non-dominated, and at least one solution in the previous front dominates it. Unlike regular sorting, where each element is assigned a different position, in NDS, multiple solutions may be assigned to a single front. The **Rank** of a solution is defined as the index of the Front it belongs in. The solutions that are not dominated by any other solution in the population set

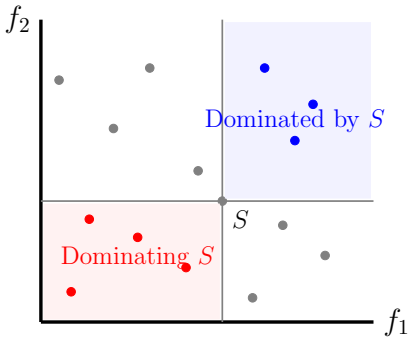


Figure 7.1: Dominance relation between multi-objective solutions: The solutions highlighted in red are dominating solution S, as they are better in both objective values f_1 and f_2. Similarly, solutions in blue are dominated by solution P since both their objective values are worse than that of S, and the ones that are not highlighted are non-dominated as they are better in one of the objectives but worse in the other.

(rank 1) constitute the Pareto-Optimal Front.

7.3.2 Related Work

The Corner Sort algorithm was proposed by Wang and Yao in 2013 [295]. Corner Sort is an NDS algorithm that focuses on finding a single non-dominated solution (called the 'corner solution' [296]) and using it to find the group of solutions that are dominated by it. We then ignore this group of solutions for the current rank assignment since these solutions cannot be in the same front as the corner solution. The number of comparisons between two solutions of many-objective optimization problems is very large. Corner Sort has the same time complexity as other common algorithms but works better than most previous algorithms as it saves the number of comparisons made even for many-objective problems.

We start every iteration of Corner Sort by finding a solution among the current set that is non-dominated with every other solution. The solution with the best value of any objective (say, the j^{th} objective) will automatically be non-dominated with every other solution of the set [**From the definition of Dominance**]. We refer to this solution as the 'corner solution' for the j^{th} objective. We find and use the corner solution to mark the group of solutions dominated by it. The dominated solutions are then ignored. The remaining solutions are checked $\forall j \in [1, m]$ repeatedly until all solutions are either marked or selected as a corner solution. The Time complexity for Corner Sort is $O(mN^2)$, where m is the no of objectives, and N is the size of the population.

The main feature of Corner Sort is that it saves comparisons by using a Non-dominated solution to mark & ignore the solutions that it dominates while assigning the current rank. It can be seen that the solution having the 'best' objective value (minimum in our case) will never be dominated by any other solution since there is at least one objective value that is always less than the corresponding objective of the

other solutions. So instead of finding all dominance relations in the solution set, Corner Sort finds solutions with the best objective value, marks it, and uses it to ignore the solutions that are dominated by it, since they won't be placed in the same rank as the solution with the best objective.

Now the same process is repeated for the next objectives till all the solutions are marked. Now we have some non-dominated solutions, and the rest, solutions dominated by at least one of the obtained non-dominated ones. The non-dominated solutions are assigned the current rank, and the dominated solutions are unmarked. Now taking the unmarked solutions as the new solution set, we repeat the whole process for assigning the next rank. In this way, all the solutions get sorted into ranks based on the Pareto dominance rule.

The implementation makes use of linked lists to store the ranked and marked lists since they allow $O(1)$ operations for removing/inserting a node. The node structures representing the solutions stored their indices, their ranks, marked statuses as well as pointers to the next/previous node in the list. In this way, the node for every solution had information about which one is the next unmarked or unranked solutions as well as the previous one. Two main functions were used, one for finding the minimum objective

Algorithm 7.1: Finding Corner solution

Data: Population Set P, number of solutions N, Current objective j,

Result: Index of solution having minimum value for j^{th} objective, idx

```
min ← INF
for i = 1 : N do
    if P^i is unmarked, and min < P^i_j then
        min ← P^i_j
        idx ← i
    end
end
return idx
```

Algorithm 7.2: Finding solutions dominated by the corner solution

Data: Population Set P, number of objectives m, index of corner solution idx

Result: Population P with solutions dominated by corner solutions marked

```
1  foreach i, such that P^i is unmarked do
2      flag ← 0
3      for j = 1 : m do
4          if P_j^i < P_j^idx then
5              flag ← 1
6              break
7          end
8      end
9      if flag == 0 then
10         mark i
11     end
12 end
```

value. Algorithm 7.1 compares the current objective values among the unmarked nodes, and returns the index of the solution with the minimum value for that objective, i.e., the corner solution. The other function uses this corner solution to find the solutions dominated by it and mark them (Algorithm 7.2). Figure 7.2 shows the working of the Corner Sort algorithm.

7.4 Methodology

7.4.1 Identifying Areas with Scope of Parallelism

The first step in achieving the required parallelism was to identify which sections of the algorithm involved a high number of independent computations that could execute concurrently on multiple threads. These sections are typically identified if they have the following characteristics:

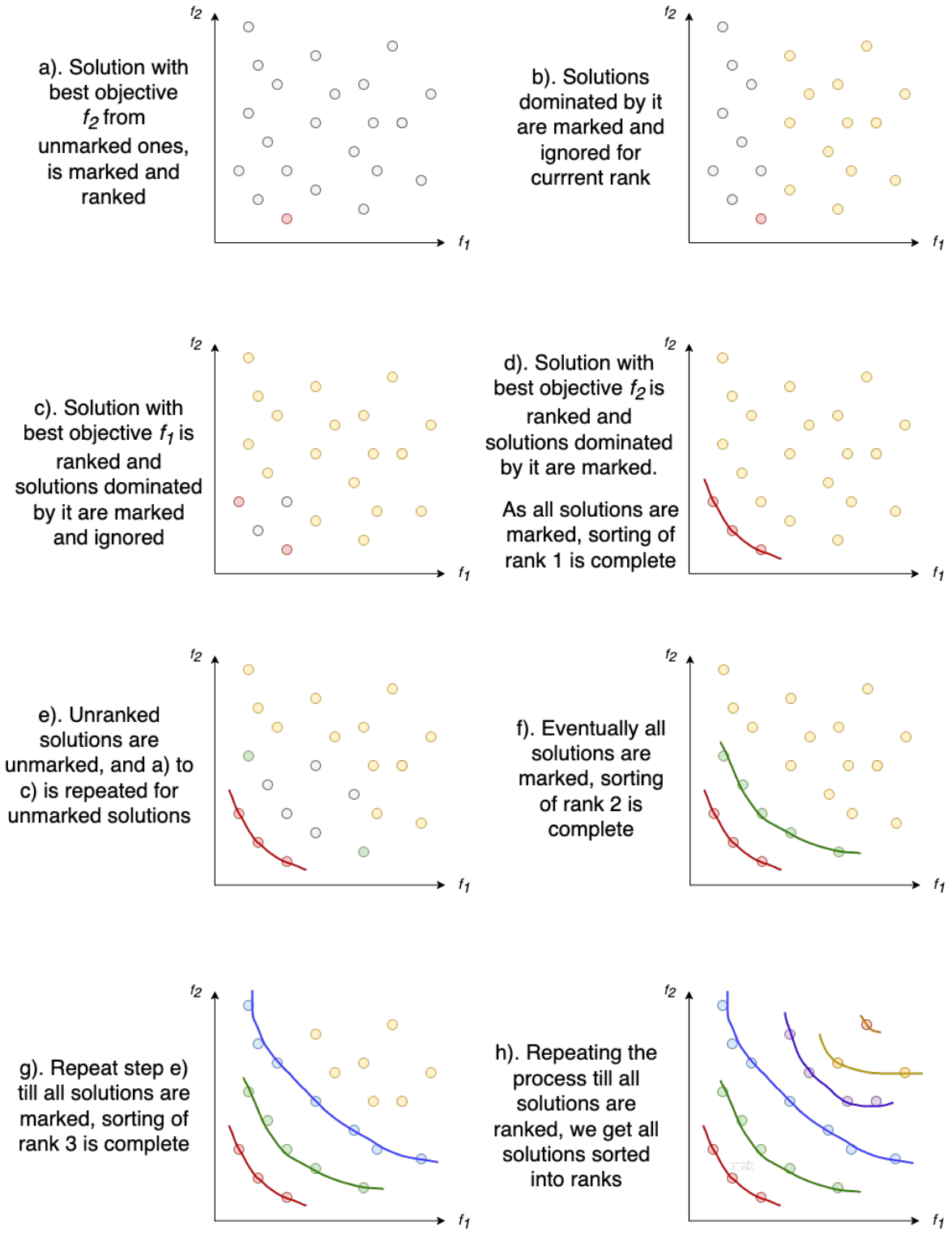


Figure 7.2: Graphical representation of Corner Sort algorithm

1. A large number of similar calculations occur sequentially. This eases the synchronization of the GPU threads (SIMT execution model for parallel computing).

2. The result of the calculation as a whole is unaffected by the ordering of the individual calculations, i.e., the result of one calculation is independent of the others.

3. The individual calculations themselves consist of only a small number of simple arithmetic operations.

On thorough analysis of the algorithm, we found two areas with the highest scope of parallelism.

7.4.2 Parallel Implementation

7.4.2.1 Finding corner solution

The first area with a high scope of parallelization is the step in which the corner solution is found (Algorithm 7.1).

For the j^{th} objective, the solution with a minimum value of f_j is found. Hence, out of N numbers and some $i \in N$, f_j^i is found, where

$$f_j^i = \texttt{MIN}(f_j^1, f_j^2, \cdots, f_j^N)$$

The time complexity of this step is $O(N)$ with $N-1$ comparisons.

It has been observed from Algorithm 7.1 that this step can be repeated many times. Comparison between two static values is a simple machine instruction independent of other factors. Thus, this step satisfies the criteria for parallelism. The minimization step is approached in a divide-and-conquer manner. The overall computation happens in two parts. In the first part, we divide the N values into smaller groups of, say, x values each. There will be a total of $\lceil N/x \rceil$ such groups. Hence, we make $\lceil N/x \rceil$ threads; each thread will find the minimum for their corresponding group of x values.

Algorithm 7.3: Parallel Function for finding corner solution

Data: Population Set P, number of solutions N, Current objective obj, current group no j

Result: Array with local minimas of $\lceil N/\log N \rceil$ parts of N obj^{th} values, $Temp[]$

```
1 min[j] ← INF
2 for i = j log N : min(N, (j + 1) log N) do
3     if P^i is unmarked, and min < P^i_j then
4         min[j] ← P^i_j
5         Temp[j] ← i
6     end
7 end
8 return Temp[]
```

The first thread finds the minimum among the values with an index from 1 to x. The second thread finds the minimum from $x + 1$ to $2x$, and so on. The last thread will find the minimum of values with an index up to N. The time complexity for each thread will be $O(x)$. Next, we use the local minima values returned by $\lceil N/x \rceil$ threads and find the global minima. The complexity of this step will be $O(\lceil N/x \rceil)$. This way, we will get the same answer as the serial method, with effective time complexity $O(x)$ + $O(\lceil N/x \rceil)$. We have taken x to be equal to $\log N$ for our implementation (Algorithm 7.3). Figure 7.3 shows the working of the parallel minimization.

7.4.2.2 Finding dominated solutions

The second area with a high scope of parallelism is where we iteratively find the solutions dominated by the corner solution found in the last step (Algorithm 7.2). In this step, we mark the solutions that are dominated by the corner solution. Finding the dominance relation between any two given solutions with m objectives takes $O(m)$ time. These objective comparisons are, again, independent of each other due to their static nature. The order in which these comparisons are made also does not matter. Thus, these

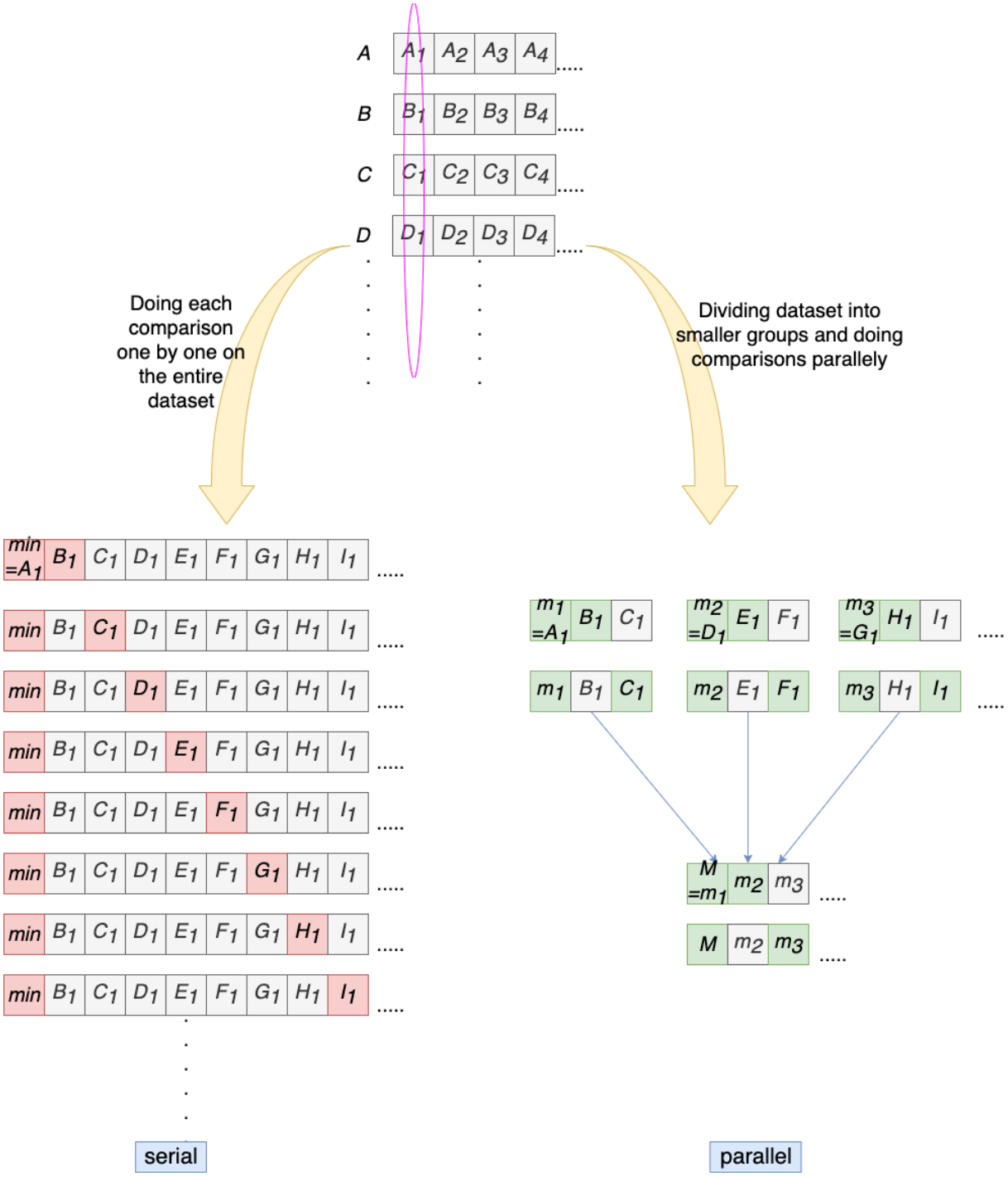


Figure 7.3: 1st Parallelization: Instead of finding the minimum objective value among all solutions simultaneously, we break it into many segments, each of whose respective minima is calculated parallelly which are used to compute the final minima.

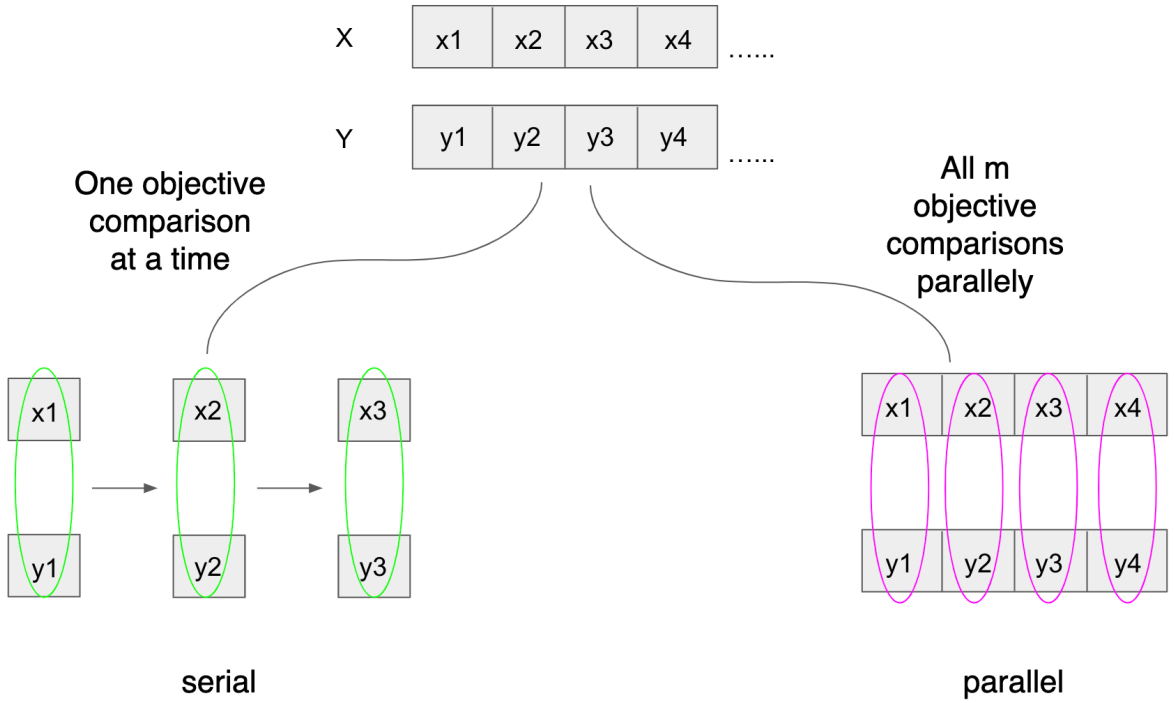


Figure 7.4: 2nd Parallelization: Solutions are checked for dominance relation parallelly on different threads rather than iteratively since they are independent of each other.

comparisons can be performed simultaneously.

We have m objective comparisons between the corner solution and the current solution whose dominance relation we have to check. Since the value of m is not as nearly as high as that of N (it is around 5-50 for most practical applications), we can distribute one objective comparison per thread. We create a boolean flag and set its value to `True`. All threads have access to this flag. The j^{th} thread will set the flag to `False` if the value of the current solution is less than the value of the corner solution for the j^{th} objective. In the end, if the flag is set to `True`, the current solution is marked (Algorithm 7.4). Figure 7.4 shows the working of the parallel objective comparisons.

7.5 Theoretical Analysis

For the general analysis of the Parallel Corner Sort, we will ignore the time taken for memory allocation/deallocation of host/device variables, and the memory transfer from

Algorithm 7.4: Parallel function for Finding solutions dominated by the corner solution

Data: Population Set P, index of corner solution idx, current objective j

Result: Population P with solutions dominated by corner solutions marked

```
1  foreach i ∈ 1 : N, such that P^i is unmarked do
2      flag ← 0
3      if P^i_j < P^idx_j then
4          flag ← 1
5          break
6      end
7      if flag == 0 then
8          mark i
9      end
10 end
```

host to device and vice-versa, as well as any input/output or pre-processing done before starting the sorting process; as was done for calculating the computation time.

The steps responsible for the complexity of the parallel algorithm are:

1. Unmarking the unranked solutions
2. Updating the rank/objective/list head variables after every rank is sorted
3. Finding $N/\log N$ best solutions parallelly
4. Finding the best out of $N/\log N$ solutions serially
5. Ranking and marking the best solution
6. Comparing best solutions with unmarked ones parallelly
7. Marking the dominated solutions

Steps 2 and 5 take constant time, so they can be ignored in our calculations.

The two main loops in the code include one running until all solutions are ranked (steps 1-7) and one that runs until all solutions are marked (steps 3-7). The first loop goes to the next iteration when calculations for the current rank/front are over. Hence it runs as many times as the number of ranks in the dataset. The second loop runs until all solutions are marked, which happens when the calculation of the current rank finishes. Since in one iteration of this loop, only a single solution is ranked (the corner solution), hence this loop runs as many times as there are solutions in the current rank. Using this information and the known time complexities of some steps, we can write an expression (without coefficients) that loosely represents the function for the time complexity of Parallel Corner Sort. The total time taken can thus be expressed in the form of the following equation:

$$T(N,m) \quad = \quad \sum_{r=1}^{R}\left(n_{ur} + \sum_{P[i]}^{i\in r}\left(\log N + \frac{N}{\log N} + n_{um}\right)\right) \quad + \quad k \quad (7.1)$$

Here, $T(N,m)$ is the computation time for N solutions with m objectives, R is the total number of ranks, n_{ur} and n_{um} are the number of unranked and unmarked solutions at any given time, and $P[i]$ are the solutions belonging in rank r, k is some constant value. The constant coefficients are ignored for simplicity.

We see that the terms $\log N$ and $N/\log N$ are constant values for a given value of N. So they come out of the second summation. Also, the number of unranked solutions, n_{ur}, is equal to the difference between the total number of solutions and the total number of ranked solutions. Hence $T(N,m)$ now becomes:

$$\sum_{r=1}^{R}\left(N - \sum_{\alpha=1}^{r} n_\alpha\right) \quad + \quad \left(\log N + \frac{N}{\log N}\right)\sum_{r=1}^{R} n_r \quad + \quad \sum_{r=1}^{R}\sum_{P[i]}^{i\in r} n_{um} \quad + \quad k \quad (7.2)$$

where n_r is the number of solutions in a given rank r ($n_r = \sum_{P[i]}^{i\in r} 1$). The second

summation is simply the number of solutions in rank r, summed over all ranks, which equals to all the solutions, N. Hence the equation becomes:

$$NR - \sum_{r=1}^{R}\sum_{\alpha=1}^{r} n_\alpha + \left(\log N + \frac{N}{\log N}\right) \times N + \sum_{r=1}^{R}\sum_{P[i]}^{i\in r}(n_{um}) + k \tag{7.3}$$

Number of unmarked solutions can be written as the total number of solutions minus the ranked solutions minus the solutions dominated till now. Hence the equation becomes,

$$NR - \sum_{r=1}^{R}\sum_{\alpha=1}^{r} n_\alpha + N\log N + \frac{N^2}{\log N} + \sum_{r=1}^{R}\sum_{P[i]}^{i\in r}\left(N - \sum_{\alpha=1}^{r} n_\alpha - \bigcup_{j=0}^{i-1} dom_j^r\right) + k \tag{7.4}$$

where dom_j^r is the set of solutions dominated by solution j in rank r. Now we have two terms, n_α and dom_j^r, whose summations can only be evaluated if we know the exact distribution of the solution set. Hence the exact expression of time complexity depends on the nature of the dataset and not only the values of N and m; hence it will be different for different cases. However, by making use of some assumptions, we can simplify the above equation.

We can consider an ideal case where the ranks are uniformly distributed, i.e., each rank has the same number of solutions; hence n_r is constant. Hence for R ranks, we will have, $n_r = \frac{N}{R}$. Also, we assume that all solutions in each rank dominate all solutions in lower ranks, so the number of unmarked solutions will be the number of unranked solutions for the first element in the rank and the number of solutions in rank r remaining for the rest. Hence we get,

$$NR - \sum_{r=1}^{R}\sum_{\alpha=1}^{r}\frac{N}{R} + N\log N + \frac{N^2}{\log N} + \sum_{r=1}^{R}\left(N - \sum_{\alpha=1}^{r}\frac{N}{R} + \sum_{j=1}^{N/R}\left(\frac{N}{R} - j\right)\right) + k$$

$$= NR - \sum_{r=1}^{R} \frac{rN}{R} + N\log N + \frac{N^2}{\log N} + \sum_{r=1}^{R} \left(N - \frac{rN}{R} + \frac{N^2}{R^2} - \frac{N}{R} \left(\frac{\frac{N}{R}+1}{2} \right) \right) + k$$

$$= NR - \frac{N(R+1)}{2} + N\log N + \frac{N^2}{\log N} + \sum_{r=1}^{R} \left(N - \frac{rN}{R} + \frac{N^2}{2R^2} - \frac{N}{2R} \right) + k$$

$$= NR - \frac{N(R+1)}{2} + N\log N + \frac{N^2}{\log N} + NR - N + \frac{N^2}{2R} - \frac{N(R+1)}{2} + k$$

$$= N(R - \frac{3}{2}) + N\log N + \frac{N^2}{\log N} + \frac{N^2}{2R} + k \tag{7.5}$$

The above equation has a minima at $R = \sqrt{N}$. And at extreme values on R, it is strictly increasing. Hence the worst case arises for $R = N$ or $R = 1$ and the best for $R = \sqrt{N}$.

Best and worst case scenarios, respectively:

$$T(N,m) = \frac{3}{2}N\sqrt{N} + N\log N + \frac{N^2}{\log N} - \frac{3N}{2} + k = O\left(\frac{N^2}{\log N}\right) \tag{7.6}$$

$$T(N,m) = N^2 + N\log N + \frac{N^2}{\log N} - N + k = O(N^2) \tag{7.7}$$

It is important to note that we did this analysis for the ideal case where the population is equally divided among the ranks, and all solutions of a rank dominate all solutions of succeeding ranks. In reality, however, such an assumption would not hold. Also, in a practical scenario, the number of ranks would be dependent on both N and m. It is simple to imagine the dependency on N. For a randomly generated population,

more solutions would require more ranks to contain them. For the dependency on m, one can think that the more number of objectives there are, the more likely it becomes for a dominance comparison to fail; hence fewer solutions will be dominated each time, and each rank will contain more solutions, reducing the number of ranks. Substituting R by $R(N, m)$ in equation 7.4 along with adding constant coefficients to all variables would give a more appropriate expression for computation time and complexity. Hence, experimental results vary with the ones discussed, as we will see in Section 7.6.

7.6 Experimental Results

7.6.1 Experimental Setup and Performance Metrics

The compute setup was 1x GPU computing node. We utilized 1x Nvidia Tesla v100 GPU and Intel Xeon Skylake processor. Maximum CPU memory was set to 192 GBs, and maximum GPU memory was set to 16 GB HBM2. Linux CentOS 7.6 distro was utilized as the OS. All code was compiled using Nvidia's CUDA v11.0 NVCC Compiler and GCC v7.5.0.

The performance of the CUDA C++ implementation of Parallel Corner Sort was measured against the C++ implementation of the serial algorithm for the large population size dataset created artificially, which is described in the following sections along with their respective experiments. There were two metrics used -

1. First is *Computation Time*, which is different from execution time, as we ignore the GPU allocation and memory transfer overheads, and some other things like I/O, some variable declaration, etc., to focus on the main part of the code. It is a slightly better metric than the execution time of a code for a better understanding of the behavior of the algorithm.

2. We also compare the two implementations directly by measuring a speedup factor,

Speedup, which measures how many times faster the parallel version of Corner Sort is in comparison to the serial one. It is given by:

$$Speedup = \frac{T^c_{seq}}{T^c_{par}}$$

Where T^c_{seq} and T^c_{par} are the computation times for Serial Corner Sort and Parallel Corner Sort, respectively. Consequently, a speedup of less than 1 suggests that the serial approach is faster than the parallel approach and vice-versa.

7.6.2 Results and Discussion

For performing the comparison between serial and parallel Cornersort on large population sizes, the data had to be artificially generated. Population size as large as 200000 solutions was used for the experiments. Random numbers between 0 and 1 are generated which are reshaped according to the required N and m values of the required dataset. The pseudo code for random data generation is given in Algorithm 7.5.

Algorithm 7.5: Generating Randomized dataset

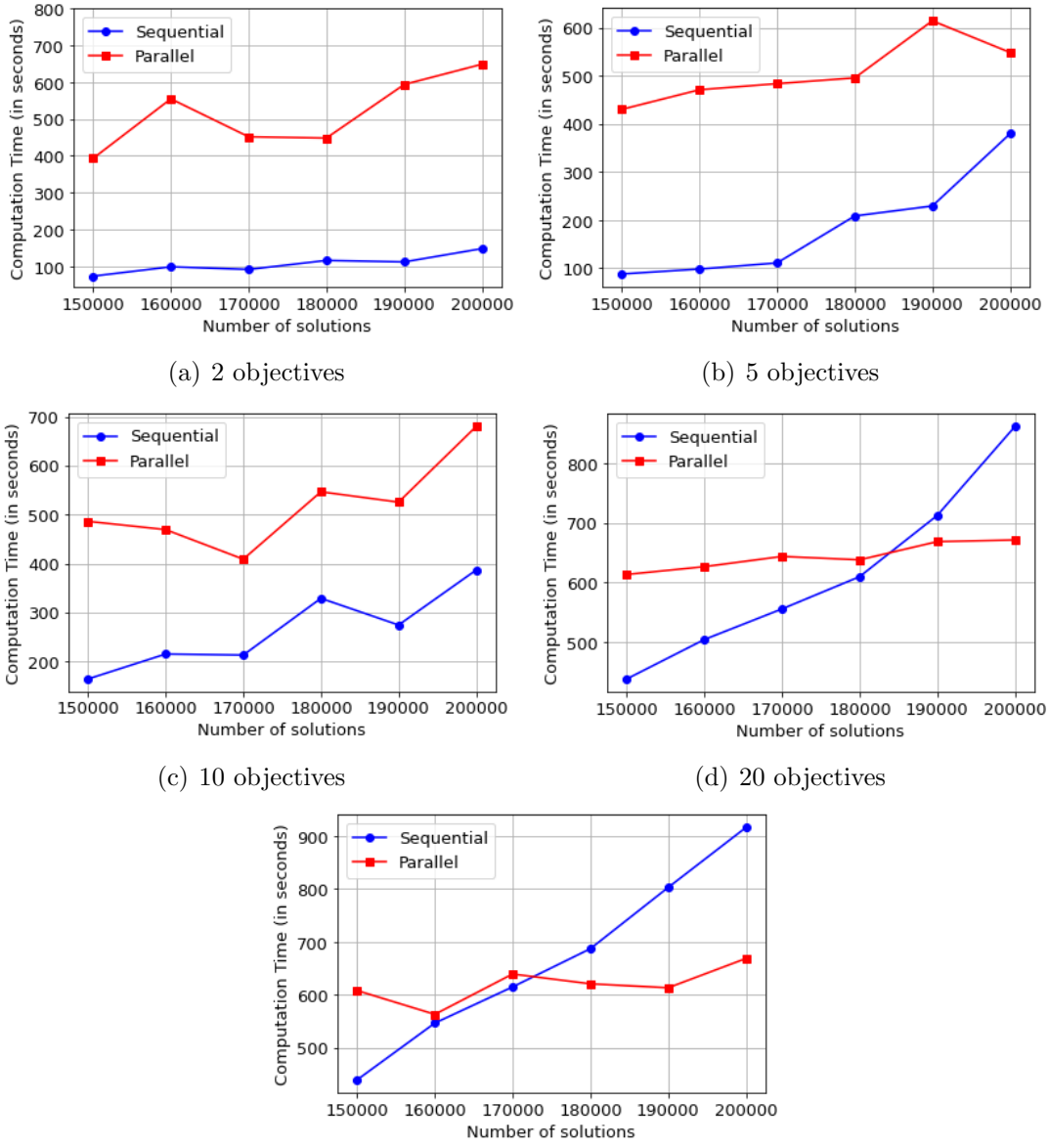
Data: Number of solutions N, Number of objectives m

Result: Dataset f with N solutions having m objectives each

```
k ← []
U(0,1) ← random number between 0 and 1
for i = 1 : N * m do
    k ← k ⋃ U(0,1)
end
k ← k(N, m)
for x ∈ k do
    f.write(x)
end
return f
```

The analysis is performed by varying the number for solutions, N from 150000 to 200000 in steps of 10000 and keeping the value of number of objectives constant, $m \in \{2, 5, 10, 20, 25\}$.

Another analysis is performed by varying the number of objectives, m to $2, 5, 10, 20, 25$ and the number of solutions constant, $N \in \{150000, 160000, 170000, 180000, 190000, 200000\}$.



(a) 2 objectives

(b) 5 objectives

(c) 10 objectives

(d) 20 objectives

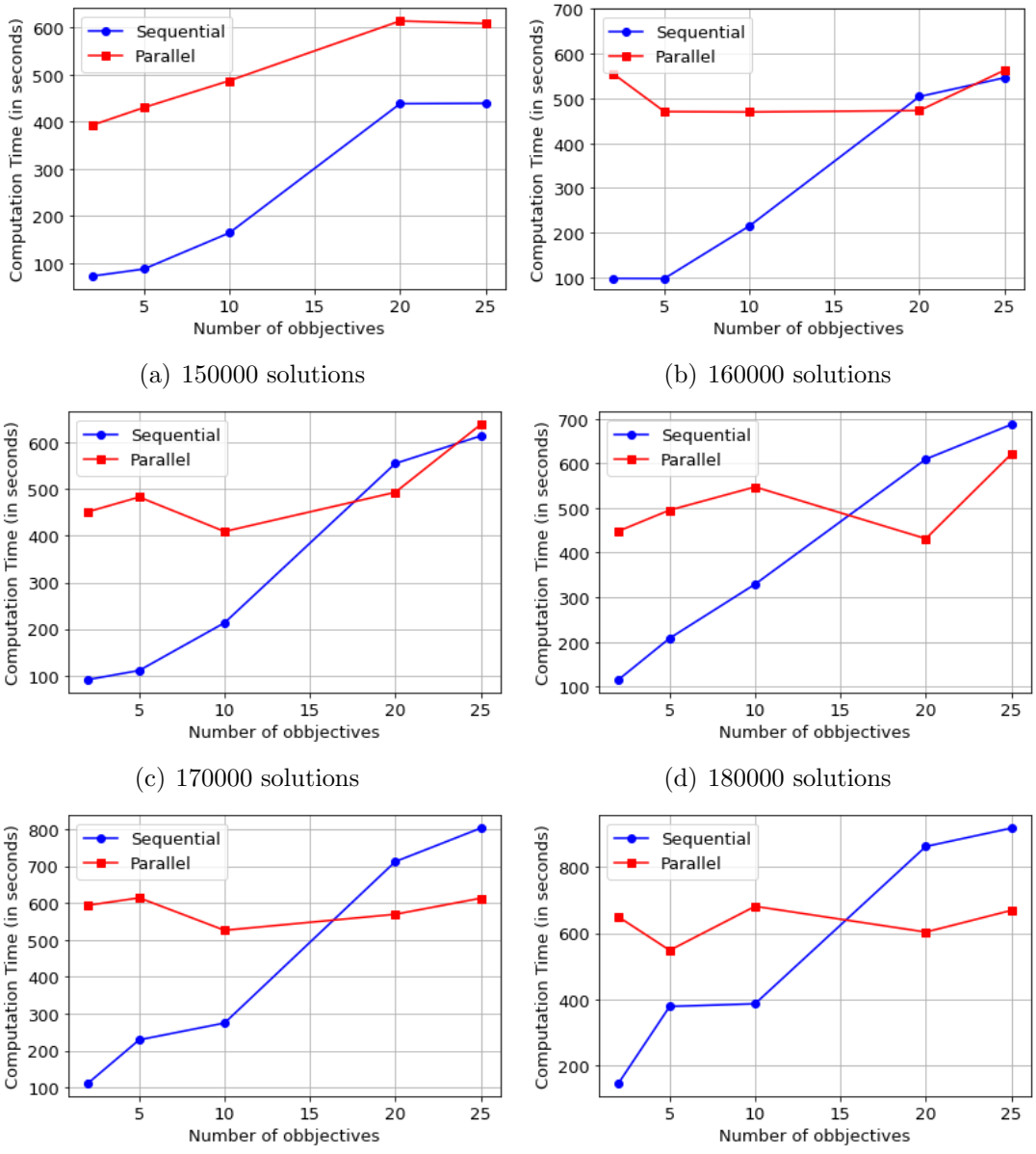
(e) 25 objectives

Figure 7.5: Comparison of Sequential and Parallel versions of Cornersort for randomly generated data, with fixed number of objectives

7.6.3 Experimental Analysis

The results obtained for the experiment on the randomized dataset can be summarized as follows.

The time required by the parallel implementation is higher than the serial implementation for less number of objectives(less than 20). We see in Fig. 7.5 that for $m = 20$



(a) 150000 solutions

(b) 160000 solutions

(c) 170000 solutions

(d) 180000 solutions

(e) 190000 solutions

(f) 200000 solutions

Figure 7.6: Comparison of Sequential and Parallel versions of Cornersort for randomly generated data, with fixed number of solutions

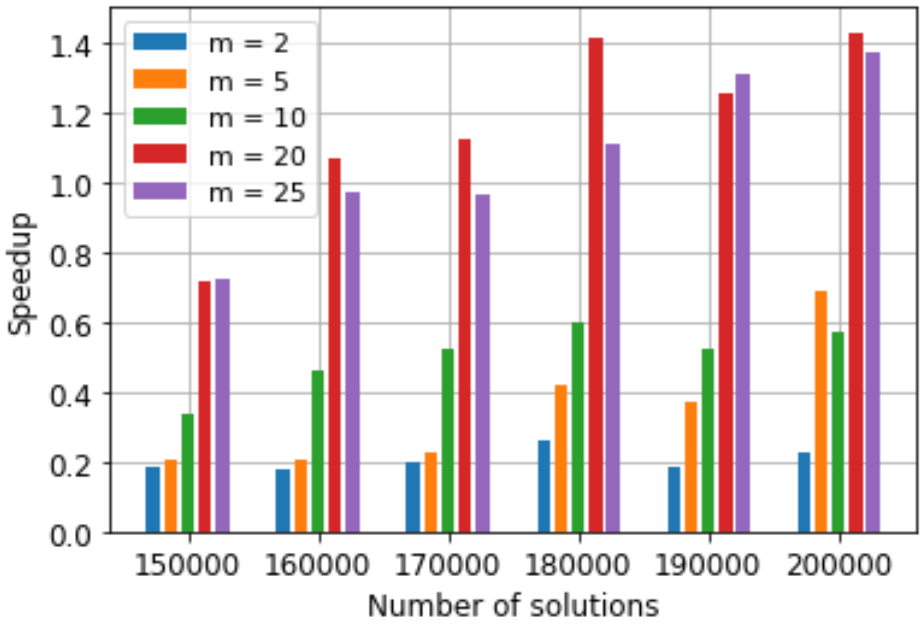


Figure 7.7: Speedup analysis of Corner Sort on randomly generated data

and $m = 25$, the serial line overtakes the parallel line after a certain number of solutions N. Hence for a high number of solutions and a high number of objectives, Parallel Corner Sort is faster than Serial Corner Sort. The point at which Parallel Corner Sort becomes faster than Serial Corner Sort also reduces from $m = 20$ to $m = 25$ objectives. This suggests that Parallel Corner Sort becomes increasingly efficient for higher number of objectives m.

Time taken also increases with the number of objectives m. We can see that the parallel and serial lines are far apart for $N = 150000$, and the parallel line intersects the serial one for all other cases between 15 and 20 objectives. For $N = 160000$ and $N = 170000$, the point of overtaking is near 20 objectives. Whereas for later cases of $N = 180000, 190000$ and 200000, it is near 20 objectives. This suggests that the Parallel Corner Sort becomes faster for larger cases than Serial Corner Sort and that Parallel Corner Sort becomes increasingly efficient as we increase the number of solutions N. (Fig. 7.6)

The speedup increases with N. In Fig. 7.7 the speedup is highest for $m = 20, 25$.

It can be seen that after 160000-170000 solutions, the speedup is larger than 1 for $m = 20, 25$ which coincides with previous results of Parallel Corner Sort being faster than Serial Corner Sort for large datasets.

7.7 Summary

In this chapter, a parallel implementation of Corner Sort has been proposed. With the help of CUDA, large sequential processes were distributed on multiple threads to achieve a lesser time complexity. For small population sizes, the parallel algorithm was slower than the serial one but eventually (for $N > 160000$), it became faster with an increase in population size, suggesting that they may be useful for large-scale applications involving big data sets. The speedup increases with an increase in the number of solutions. The rate of increase of computation time is also lower for Parallel Corner Sort than for the serial algorithm, suggesting that they remain suitable for use over a wide range of population sizes, making them apt for practical uses. We also observed that the experimental analysis of time complexity under the produced results is lower than for Serial Corner Sort. However, as of now, it is a good implementation for a large range of solutions and can be used for a wide range of practical applications.