Parallelization of Hybrid Multi-Objective Evolutionary Algorithm on Multi-Core Architectures

by

Zhuoran Sun

A thesis submitted to
The Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements
of the degree of

Master of Science

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba, Canada
December 2023

Thesis advisor

Dr. Parimala Thulasiraman

Author

Zhuoran Sun

Parallelization of Hybrid Multi-Objective Evolutionary Algorithm on Multi-Core Architectures

Abstract

Many real world optimization problems involve multiple conflicting objectives, constraints and parameters. Multi-objective optimization (MOO) techniques are used to solve these problems. The goal of MOO is to find a set of optimal solutions, or the Pareto optimal front. Multi-objective evolutionary algorithms are heuristics that evolve a population of candidate solutions to find the Pareto optimal front in a single run. The selection criterion used to select individuals in the population play an important role in determining the quality of the solutions. Pareto-based algorithms use the Pareto selection criterion to evolve different parts of the solution space introducing diverse solutions, but converge slowly to the optimal front. On the other hand, non-Pareto selection Criterion (NPC) algorithms converge faster to the Pareto front, but in the process eliminate other diverse solutions. To compensate for the strengths and weaknesses of PC and NPC, hybrid frameworks such as BCE (bi-criterion evolutionary) have been proposed. In BCE, the PC and NPC algorithms evolve separately, but also co-operate by exchanging information to explore and exploit the objective space. In the literature, two well-known evolutionary algorithms, Non-dominated Sorting Genetic Algorithm II (NSGA-II) (PC) and Multi-objective Evolutionary Algorithm based on Decomposition (MOEA/D) (NPC) have been used as a case study in the BCE framework. However, the individual algorithms are computationally expensive.

In this thesis, we study the parallelization of the BCE framework. NSGA-II is highly data parallel, and is well suited for single instruction multiple data architectures. MOEA/D is non-data parallel with some parts of the algorithm being sequential. Therefore, we design the parallel NSGA-II algorithm on the GPU multi-core accelerator and parallel MOEA/D algorithm on multi-core CPU machines using an island model. Using the travelling salesperson benchmark data sets we analyze the performance of the parallel hybrid algorithm quantitatively and qualitatively using metrics such as IGD scores, scalability, and speedup.

Contents

List of Figures

Acknowledgments

First of all, I would like to thank my advisor, Dr. Parimala Thulasiraman, for guiding me through the years ever since my undergraduate honours project. My gratitude is beyond words for all of your supports and advice.

I would like to thank Dr. Shaun Lui and Dr. Shahin Kamali, for being my committee members.

I would also like to thank Dr. Ying Ying Liu from IDEAS lab for giving advice on my research.

This thesis is dedicated to loved ones.

Chapter 1

Introduction

1.1 Overview

Many real-world applications can be represented as networks. Common examples include transportation, biological, social or epidemiological networks. A network consists of entities that is divided into a set of objects called nodes and a set of relationships between the nodes called links. Nodes and links are then projected onto an abstract mathematical geometric graph formation as vertices and edges, respectively. Graphs are useful data structures to model networks. The network is usually represented as a weighted graph. Real-world applications can be formulated as an optimization problem [42]. That is, optimize (maximize or minimize) an objective function with some constraints. For example, in a transportation network, the goal of the routing algorithm

is to minimize the travelling distance between source and destination. Here the the nodes represent the intersections and edges the roads. Another example, in business network application, the nodes may represent the portfolios and the edges between the portfolios. The problem is then to cluster the portfolios that maximize profit. Problems targeting a single solution with the goal of optimizing a single objective function are termed as single-objective optimization problems (SOPs). The aim of such modelling is to find an (or a set of) optimal solution(s) to the given problem.

In many real world applications, however, the optimization problem translates to finding optimal solutions by considering more than one objective at the same time. And the objectives are often conflicting with each other. For example, in vehicle routing problem, the goal is to find a route between two points that will minimize *both* the travelling time and travelling distance. Another example with competing objectives can be found in the process of designing aircrafts within aerospace industries, where the goal is to increase the robustness of the aircrafts (i.e., a maximization problem) but also to keep the manufacturing costs as low as possible (i.e., a minimization problem). Such problems are termed as *multi-objective optimization problems* (MOPs) and are increasingly

popular in a wide range of disciplines.

Multi-objective optimization problems (MOP) formulate optimization problems as either minimization or maximization problems with one or more conflicting objectives. The quality of the solution is determined by performing a test called the *dominance* test. This can be defined as follows. Given two solutions, x_1 and x_2, solution x_1 dominates solution x_2 if:

- Solution x_1 is not worse than solution x_2 in all objectives, and
- Solution x_1 is strictly better than solution x_2 in at least one objective.

If the above conditions hold, we say solution x_1 dominates solution x_2. Or in other words, solution x_2 is dominated by solution x_1. The non-dominated solution set is a set of all the solutions that are not dominated by any other member of the solution set, and the non-dominated set of the entire feasible decision space is termed as the Pareto-optimal set (PS). We use the set of all points mapped from the Pareto optimal set in the objective space to define the boundary, which is called the Pareto optimal front (PF). The goal of an algorithm in solving MOPs is to find the PS from decision space such that the PF in the objective space can be best projected. When assessing a MOP algorithm, we

focus on the optimality and diversity of its solution. Optimality means that each solution in the set is a well representation of an optimal combination of the competing objectives. Diversity means that the points on PF mapped from the solutions in the set are evenly distributed [45]. Finding diverse optimal solutions for most multi-objective optimization problems is an NP-Hard problem. There is no polynomial time algorithm and heuristics are used to solve MOP [42]. *In this thesis, we consider evolutionary multi-objective optimization (EMO) heuristics.*

EMO heuristics evolve a population of candidate solutions to find the Pareto optimal front in a single run. The selection criterion [32] used to select individuals in the population play an important role in determining the quality of the solutions. Pareto-based algorithms use the Pareto selection criterion to evolve different parts of the solution space introducing diverse solutions, but converge slowly to the optimal front. On the other hand, non-Pareto selection Criterion (NPC) algorithms converge faster to the Pareto front, but in the process eliminate other diverse solutions. To compensate for the strengths and weaknesses of PC and NPC, hybrid frameworks such as BCE (bi-criterion evolutionary) have been proposed.

In BCE, the PC and NPC algorithms evolve separately, but also

co-operate by exchanging information to explore and exploit the objective space. In the literature, two well-known evolutionary algorithms, Non-dominated Sorting Genetic Algorithm II (NSGA-II) (PC) [17] and Multi-objective Evolutionary Algorithm based on Decomposition (MOEA/D) (NPC) [56] have been used as a case study in the BCE framework [32]. MOEA/D decomposes a MOP into a set of scalar optimization sub-problems, and solves them simultaneously to provide an aggregation as the final solution to the MOP.

One variation of MOEA/D with respect to parallelization is proposed in [38]. In this work, an island model is used for solving each subproblem simultaneously after decomposition. In other words, after applying decomposition, each subproblem is represented by a single island. Neighbourhood is then defined for each island. And islands evolve independently in parallel, with periodical exchanges of information between an island and its neighbouring islands. More specifically, good solutions from an island's neighbouring islands are used to replace the island's bad solutions. In [38], islands exchange solutions asynchronously. *In this thesis, we develop a variation of parallel MOEA/D with island model, with different update mechanism and synchrounous communication.*

The individual algorithms, however, are computationally expensive. *In this thesis, we study the parallelization of the BCE framework.*

The sequential BCE framework proposed by Liu et al. [37] is used as the basis of the proposed parallel design and implementation. NSGA-II is highly data parallel, and is well suited for single instruction multiple data architectures. MOEA/D is non-data parallel with some parts of the algorithm being sequential. Therefore, we design the parallel NSGA-II algorithm on the GPU multi-core accelerator and parallel MOEA/D algorithm on multi-threaded multi-core CPU machines. Each of the sub-problems are executed in parallel, exchanging information at the end of each generation. Using the travelling salesperson benchmark data sets we analyze the performance of the parallel hybrid algorithm quantitatively and qualitatively.

The rest of the thesis is organized as follows. Chapter 2 and Chapter 3 provide a brief background on what is required in understanding this thesis. Chapter 4 presents the literature review on evolutionary algorithms. Our proposed parallel approach is described in Chapter 5. Experiments and results are presented in Chapter 6. Chapter 7 summarizes the thesis and discusses potential future work.

1.2 Contribution

The contribution of this thesis is to study the parallelization of the hybrid algorithm for solving MOP problems. We choose the multi-core architecture that best suits the individual algorithms for increased performance. We test the parallel algorithm against the sequential algorithm for the correctness of the results using benchmarks from multi-objective travelling salesperson problem data sets and provide performance analysis using standard parallel computing metrics.

Chapter 2

Background on Multi-Objective Evolutionary Algorithms

Evolutionary algorithms (EA) are inspired by the Darwinian principle: selection of the fittest. Genetic algorithm (GA) is an example of EA. GA maintains a population and using crossover and mutation, evolves new population of candidate solutions in each generation. EAs are promising heuristics for solving optimization problems. They are highly adaptive to uncertainties in complex problems, simple to understand and can be flexibly applied to real world problems. Moreover, population-based techniques such as GA they are easily parallelizable.

Over the past few years, evolutionary algorithms have been studied to solve MOP. These algorithms are categorized into evolutionary multi-objective optimization algorithms.

2.1 Evolutionary Multi-objective Optimization Algorithms

An evolutionary multi-objective optimization algorithm (EMO) algorithm manipulates a population of solutions in every iteration and is therefore naturally suited for finding a set of non-dominated solutions in MOPs. Although an EMO algorithm does not guarantee to find the Pareto optimal solutions as it is only a heuristic, it tries to find sub-optimal solutions through repeated improvements to non-dominated solutions. Therefore, when solving practical problems, an EMO algorithm finds solutions based on two principles [18]:

- Find the non-dominated solutions with sufficiently good optimality and high diversity.
- Decide on a final solution by using higher-level information.

An important advantage of using an EMO algorithm compared to traditional posteriori techniques for solving MOP is that the algorithm can return multiple candidate solutions (i.e., multiple non-dominated points) which indicate different trade-off within a single simulation run [18]. The good solutions (closer to Pareto front) are inserted into the population pool to force the algorithm to explore the area that produces

the good solution. Through reproduction operators, the algorithm introduces new solutions directed towards the Pareto front.

EMO algorithms can be categorized into two categories based on their evolution process and selection criterion: Pareto dominance EMO, or referred to as Pareto Criterion (PC) evolution, and Non-Pareto Criterion (NPC) evolution [37].

PC Evolution: PC evolution techniques emphasize Pareto dominance when selecting offspring in each generation. A vector is used to store the fitness scores of each individual in the population and the individuals are ranked based on their Pareto dominance. In a MOP, the final solution is a set Pareto non-dominated solutions. Therefore, it seems reasonable the individual selection is based on Pareto dominance. However, the drawback of only focusing on Pareto dominance when selecting individuals is that it may fail to distinguish between individuals when each of the individual solutions have their own advantage in different objectives [32]. In general, PC evolution techniques suffer from slow convergence to optimal front [48], lack of indication and interpretation on the quantitative difference between the two individuals for different objectives [12], and performance being less satisfactory if the Pareto set has complex characteristics (i.e., if the MOP involves multiple trade-off,

non-linear dependencies, or conflicting objectives) [31]. *In this thesis, NSGA-II is used as the algorithm for prompting PC evolution.*

NPC Evolution: Methods in NPC evolution categories are based on decomposition techniques in solving a MOP. An NPC method decomposes a given MOP into a number of scalar, optimizing sub-problems. Each sub-problem has its own population maintained for its own evolution. A real value representing the fitness score is assigned to each individual in this population, and individuals are ranked by this score when selecting the fittest one as the final optimal solution to this specific sub-problem. An aggregation of all the optimal solutions from each sub-problem is the final output to the MOP. By using decomposition strategy, NPC evolution prompts higher selection pressure towards the Pareto front and thus faster convergence [27]. Although it is not being used in this proposed work, another advantage of NPC evolution is the possibility of embedding local search techniques to further facilitate search of good solution for each sub-problem [12] [31]. The idea of pushing higher selection pressure towards optimal front has it's own disadvantages. The final solutions may not be distributed uniformly along the Pareto front as different parts of the Pareto front may be treated differently and some Pareto optimal solutions may be dropped during the evolution

process depending on the criterion used [32]. Another challenge faced by NPC evolution is how to maintain the uniformity of points of the Pareto front [32]. Ideally, the points should be evenly distributed along the Pareto front when the set of non-dominated solutions shows good diversity. These techniques are fast at convergence but do not produce diverse solutions.

In this thesis, MOEA/D technique is used to lead the NPC evolution process.

2.2 Non-dominated Sorting Genetic Algorithm II (NSGA-II)

Being one of the widely used EAs to solve MOPs, non-dominated sorting genetic algorithm II (NSGA-II), proposed by Deb et al. [17] stimulates the natural selection inspired by the Darwinian theory. NSGA-II becomes to the PC evolution category. As mentioned above Pareto dominance techniques fail to distinguish between individuals when each of the individual solutions have their own advantage in different objectives [32]. Therefore to alleviate this, density information around each individual in the population are used to further rank the individuals. NSGA-II [17] represents one such algorithm. Figure 2.1 gives a high

level overview of NSGA-II.

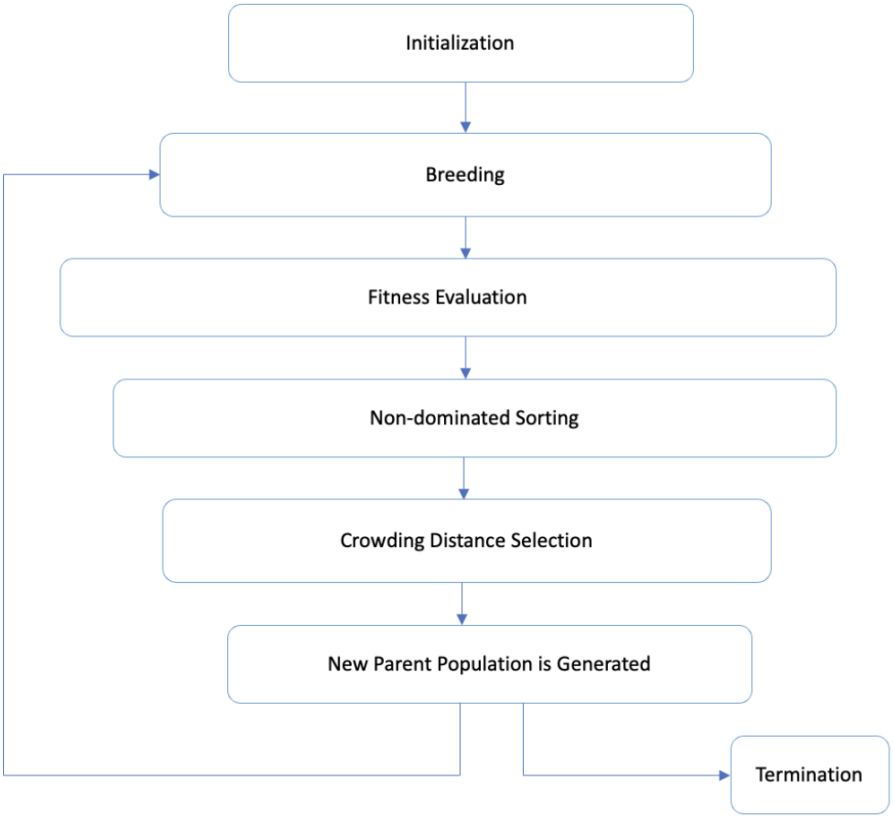

Figure 2.1: NSGA-II Overview

NSGA-II emphasis is on elitism (survival of the fittest), diversity preservation and non-dominated solutions to find multiple Pareto solutions. As the algorithm is based on non-dominated solutions, (recall Chapter 1) these solutions form the Pareto front. Given two or more conflicting objectives, non-dominated solutions provide the solutions that best fits the objectives. This implies, a solution x is compared

with every other solution in the population to determine its dominance. If none of the solutions dominate it, then solution x is a non-dominated solution and is selected by the NSGA-II. This solution will be one of the solutions in the Pareto set to form the Pareto optimal solutions. Mathematically, given a set of objectives, $f_i, i = 1, ..., n$, and two feasible solutions, x_1 and x_2, solution x_1 dominates solution x_2 if: $\forall i \in 1, 2, ..., n, f_i(x_1) \leq f_i(x_2)$ and $\exists i \in 1, 2, ..., n f_i(x_1) < f_i(x_2)$.

In this thesis, we will use NSGA-II as our algorithm to drive PC evolution process. NSGA-II follows the following steps when evolving the solutions set [17] [37]. Figure 2.2 illustrates the details:

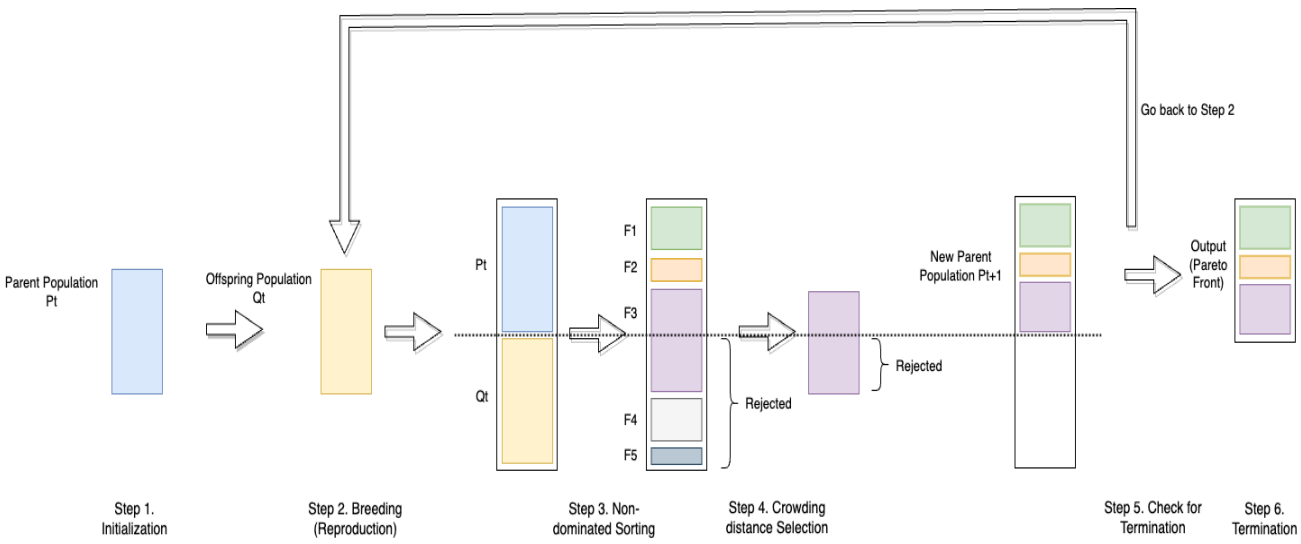

Figure 2.2: NSGA-II Algorithm Steps

1. Create Initial Population: An initial parent population P_t will be created, filled with solutions which are generated randomly. Each

solution in the population is termed as an individual to represent chromosomes from real world biological behaviours. Since each individual in the population is represented by a chromosome, it contains a string of genes. In the context of NSGA-II, our the genes will be representing the values of the decision variables. In order to determine which chromosomes (i.e., individuals in the current population) are the fittest to survive, we use objective functions to calculate a fitness value and assign it to each individual. Since we are dealing with MOPs, the fitness score for each individual will be in the form of a vector instead of a single value as in SOPs. And the size of the fitness vector will be equal to the number of objectives in the given problem.

2. Breeding (Reproduction): An offspring population Q_t will be created from parent population by a series of genetic operations. The common genetic operations include crossover and mutation. Crossover takes sections from individuals in the parent population and recombines them to form a new individual. Mutation tweaks an element (or a couple of elements, depends on the program setting) of an individual to another element and this happens with a predefined probability.

3. Whole Population: With NSGA-II, the parent and offspring populations ($P_t + Q_t$) are combined to form a whole population. The size of the whole population is $2N$.

4. Non-dominated Sorting: Non-dominated sorting is applied to this P_{t+1} by ranking and classifying all the individuals into different fronts based on their fitness vectors. In other words, the individuals are sorted according to an ascending level of non-domination. A new population is then formed by filling in with individuals based on their front rankings. The first front (PF_1)is completely non-dominant set from the current population. The second front (PF_2) being dominated by the individuals in the first front only and the front goes so on. Rank (fitness) is assigned to each individual in each front depending on which front they belong to. Individuals in the first front are assigned fitness 1, the individuals in the second front are assigned fitness 2, and so. Note that the individuals from the smallest rank (rank 1) are chosen first, then from rank 2, and so on, such that they do not exceed size N. *This process introduces elitism.*

   In Figure 2.2, F_1 and F_2 will have a place in the new population, P_{t+1}. F_4 and F_5 will be discarded by their lower ranks front. F_3

front is taken partially. Crowding distance selection (next step) will be applied to individuals in that front to choose which ones will be included in the new population. This is done to keep the size of the new population consistent.

5. Crowding distance Selection: *This process ensures diversity.* Calculate the crowding distance of each individual in the same rank. It is applied to the least optimal Pareto front (from previous step) to choose solutions with largest crowding distance to be included in the new population. In Figure 2.2, this front is front F_3. The crowding distance is a measure of how close the individual is to their neighbours. Large crowding distance results in better diversity. Individuals from different ranks are selected according to rank order. The newly created population will be the parent population for the next generation. That is, solutions in F_1, F_2 and selected solutions from F_3 will be the new population $P_t + 1$ in the next generation. At the end this step, there are N individuals in the population.

6. Check for Termination: If the terminating condition (eg., user defined number of generations) is satisfied, go to Step 6. Otherwise, go back to Step 2 and continue the process. Set $P_t = P_{t+1}$.

7. Termination: Program will stop and the final Pareto front will be returned as the output of the algorithm.

2.3 Multi-objective Evolutionary Algorithm based on Decomposition (MOEA/D)

Another well-known EA for solving MOPs is the multi-objective evolutionary algorithms with decomposition (MOEA/D), proposed by Zhang et al. [56]. MOEA/D technique belongs to the NPC evolution category. A high level description of MOEA/D is provided in Figure 2.3.

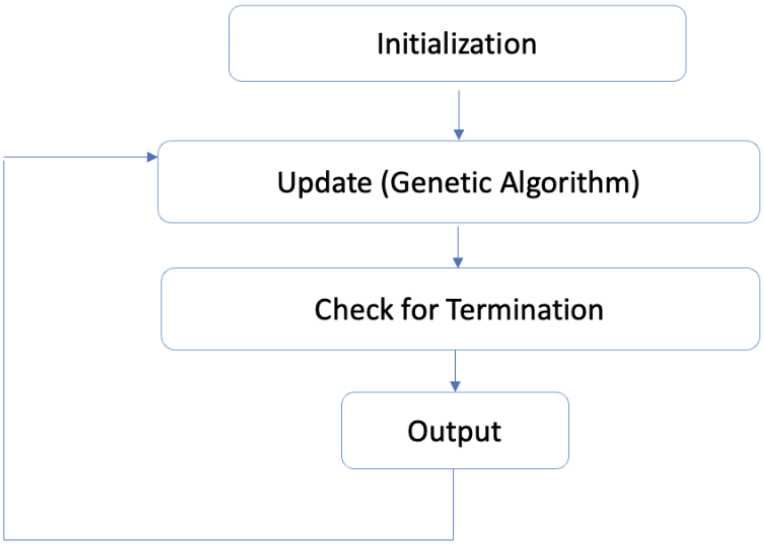

Figure 2.3: MOEA/D Overview

Unlike NSGA-II that uses Pareto dominance as the selection criteria, MOEA/D uses the decomposition strategy.

Decomposition Strategy: Rather than treating the multiple objectives as one single MOP, MOEA/D decomposes one MOP into a number of scalar optimization sub-problems explicitly. Each scalar sub-problem works on its own to find one optimal solution to its *assigned objective function* by using a predefined search heuristic, and only information obtained from its neighbouring sub-problems in the current generation

is used during the optimizing process. A population of best solutions found so far is maintained and updated by each sub-problem. This gives MOEA/D the capability of optimizing each solution from all sub-problems simultaneously. Finally, the global solution (i.e., the final output solution to the single MOP) is an aggregation of all partial results returned by each sub-problem.

Neighbourhood Concept: Another concept used in MOEA/D is the defining a neighbourhood for each sub-problem [56]. A neighbourhood with a predefined size is assigned to each sub-problem based on the closest pair-wise Euclidean distance between the sub-problem's weight vector and other sub-problems' weight vectors. Only the current solutions of its neighbouring sub-problems are involved when optimizing each sub-problem. This allows effective and efficient *local exploitation* when solving MOP.

MOEA/D - Decomposition methods: By decomposing one MOP into a number of scalar optimization sub-problems, the approximation of the Pareto front (PF) is also decomposed and is distributed among all sub-problems. Research shows that a reasonably large number of evenly distributed weight vectors can usually lead to a set of Pareto optimal

vectors. Although the resulting Pareto optimal vectors may not spread evenly, the PF will still be very well approximated [56].

Number of methods exist to construct aggregation functions for doing decomposition. Among them, the weighted sum approach and Tchebycheff approach are the most popular ones. *In this thesis, we will use the weighted sum approach as the decomposition technique for MOEA/D implementation.*

Weighted sum approach [40]: Let λ be a weight vector such that $\lambda = (\lambda_1, ..., \lambda_m)$ with $\sum_{i=1}^{m} \lambda_i = 1$ and $\lambda_i \geq 0$ for all $i = 1, ..., m$. The decomposed sub-problems are then defined as:

$$Minimize \text{ or } Maximize \quad g(x|\lambda) = \sum_{i=1}^{m} \lambda_i f_i(x) \tag{2.1}$$

The optimal solutions to these sub optimization problems are Pareto optimal to the MOP if the PF is convex. More precisely, $g(x|\lambda)$ is used to emphasize that λ is a weight vector in that objective function, m is the number of objectives in the problem, x is the variable to be optimized or in other words it is a solution in the decision space, $\lambda \geq 0$ is the elements of the weight vector λ_i for the i_{th} subproblem, $\lambda_i f_i(x)$ is the i_{th} objective value for solution x.

The MOEA/D algorithm is described below [56]:

1. Initialization: Decompose the MOP into N sub-problems. Let P_k where $k = 1, 2, .., N$ denote population kept by each sub-problem. Initially, each population P_k randomly selects a set of solutions. Let i denote an individual that represents a solution to the i_{th} sub-problem. A neighbourhood, denoted by B_i, is then assigned to each individual i_{th} sub-problem. The size of each neighbourhood denoted by T, $T < N$ is defined by the user. For each sub-problem k, we define its T neighbours by calculating the pair-wise Euclidean distance between its weight vector and the weight vectors of other sub-problems. The T sub-problems that have the closest Euclidean distance are then said to be the neighbourhood of i_{th} sub-problem. Note for each i_{th} sub-problem, it will include itself in its neighbourhood. The algorithm then compute and store the objective value for each of the N sub-problem.

   EP, called the external population, stores the non-dominated solutions from all sub-problems selected during the search process. Initially, $EP = \emptyset$.

2. Reproduction: For each sub-problem, i, we select two other sub-problems from its neighbourhood (i.e., B_i) to be the parents of the

new solution generated by sub-problem i. Let y denote the new solution generated by performing genetic operations on the parents from B_i for sub-problem i. This process is applied to each of the N sub-problems. By the end of reproduction stage, he size of the offspring population will be the same as the parent population.

3. Improvement: To improve each newly generated child solution y, a problem-specific improvement heuristic might be applied to y to create y'. This is an optional step in the overall process. (Note: in the case of MTSP, the 2-opt local search can be applied to improve a child solution [37]. But we do not consider any improvement heuristics in this thesis as it is not efficient to use any in terms of parallelization. The use of improvement heuristics might be considered as a potential improvement for this thesis in the future works.)

4. Update Neighbourhood: Since we do not apply any improvement heuristics to our child solution y, we will use y directly in this step. (Note: If an improvement is being made from previous step, we will use the improved solution y' in this step.) In this step, solution y from sub-problem i is evaluated by i's neighbouring sub-problems based on their weight vectors. And the current solution

of i's each neighbouring sub-problem will be replaced by solution y if the resulting cost produced by computing neighbour's objective function on y is strictly better. *One of the advantages of comparing the objective costs between neighbouring sub-problems instead of Pareto optimality is that the computational costs is significantly lower, and it is more efficient to compute for larger number of objectives [37].*

5. Update EP: Recall that EP stores all the non-dominated solutions from all subproblems selected during the search process. In this step, y is used to compare with all the solutions in EP. Solutions in EP will be removed if they are dominated by y. And y will be added to EP if none of the solutions in EP dominates it.

6. Check for Termination: If the terminating condition is met, algorithm stops and returns EP as final output to the MOP. Else, go to Step 2 to continue the process.

2.4 Hybrid Algorithm: Bi-Criterion Evolution (BCE) Framework

As one of the PC evolution techniques, NSGA-II achieves a higher diversity for the final set of solution. However, the convergence speed is slow compared to MOEA/D, which represents the NPC evolution process but scores a lower diversity in final result. Recall that in this work, we will use NSGA-II for PC evolution and MOEA/D for NPC evolution, and Figure 2.4 summarizes the major advantage and disadvantage for these.

	PC Evolution (NSGA-II)	NPC Evolution (MOEA/D)
Major Advantage	Higher diversity	Faster convergence
Major Disadvantage	Slower convergence	Lower diversity

Figure 2.4: Major Advantage and Disadvantage for PC and NPC Evolutions

To take strength of both PC and NPC evolution's and to compensate for their weaknesses, the Bi-Criterion Evolution (BCE) framework [32] separates the populations and reproductive processes into two main evolution processes, and utilizes PC and NPC evolution's for each of the two processes. By allowing these tow techniques work collaboratively, BCE produces results with high diversity at a faster speed.

Figure 2.5 shows the basic concept of the BCE framework, the two

main boxes show the separate evolution process for each PC and NPC process, and the arrows in-between indicate the collaboration between them [32]. In other words, by dividing the whole process into two separate parts and letting PC and NPC evolution each be responsible for one part, the slow convergence of PC evolution is compromised by NPC evolution and the loss of diversity resulted from NPC evolution is compensated by PC evolution.

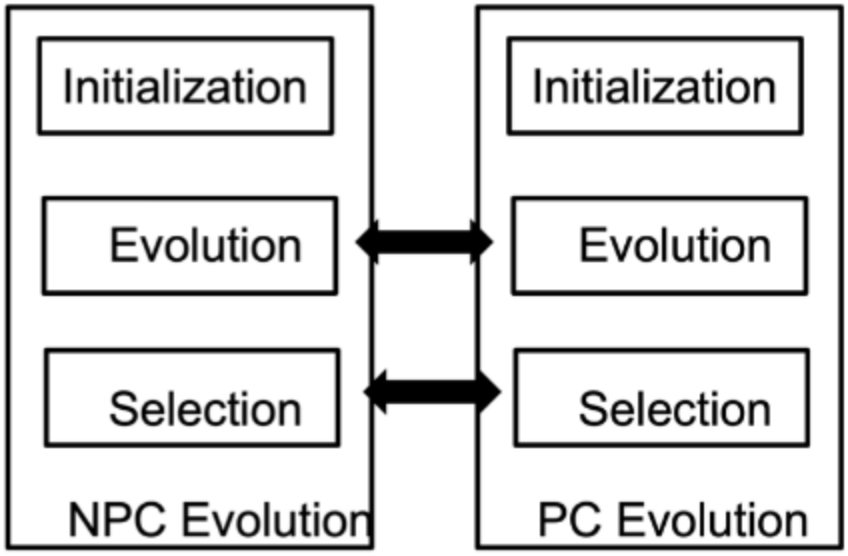

Figure 2.5: PC and NPC Processes in Original BCE Framework [32]

From Figure 2.5, we can see the BCE framework consists of two parts. From the original work in [32], the NPC evolution can take in

any non-Pareto based algorithm to drive the PC evolution forward to the optimal front. And the PC evolution performs two operations:(i) population maintenance and (ii) individual exploration.

Population maintenance maintains a set of representative non-dominated individuals and individual exploration explores potential good areas that are not yet being examined in NPC evolution. The idea of working collaboratively is an important concept in this framework, as to facilitate each other's evolution, information will be exchanged at a certain frequency between PC and NPC evolution. More specifically, each evolution will keeps a population for its own process. PC population will be responsible for higher diversity. NPC population drives the search towards optimal front at faster pace. After one population finds good individuals, the other population will get to use them in its own process by receiving them from the communication. In this framework, the authors kept the choice of algorithm for NPC evolution open and proposed a new PC evolution strategy based on individual exploration with a novel population maintenance strategy.

2.4.1 A Cooperative, Concurrent, Coevolutionary Multi-objective Optimization (CO^3MO)

Extended from the BCE framework [32], Co^3MO proposed by Liu et al. [37] is a novel hybrid framework that combines NSGA-II and MOEA/D for solving MTSP. It uses NSGA-II for PC evolution and MOEA/D for NPC evolution. The two algorithms compute independently when finding optimal individuals and coevolve by exchanging information frequently. More specifically, when updating the population for one evolution, both the current candidate solutions of itself and of the other evolution will be considered. The final output produced by the framework will be an aggregation of the two populations from both evolution's. From experiments performed on a bi-objective MTSP benchmark data set randomAB100, Liu et al. [37] found that NSGA-II can produce an evenly distributed Pareto front, but the extreme points were being neglected; on the other hand MOEA/D was able to lead the search direction towards the extreme points, but the population diversity was less desirable compared to NSGA-II. Then Liu et al. [37] combined NSGA-II and MOEA/D into Co^3MO and performed experiments on the same data set. The results indicated that both evolution processes were improved by cooperation through exchange of good so-

lutions. Figure 2.6 shows a high level program flow for Co^3MO.

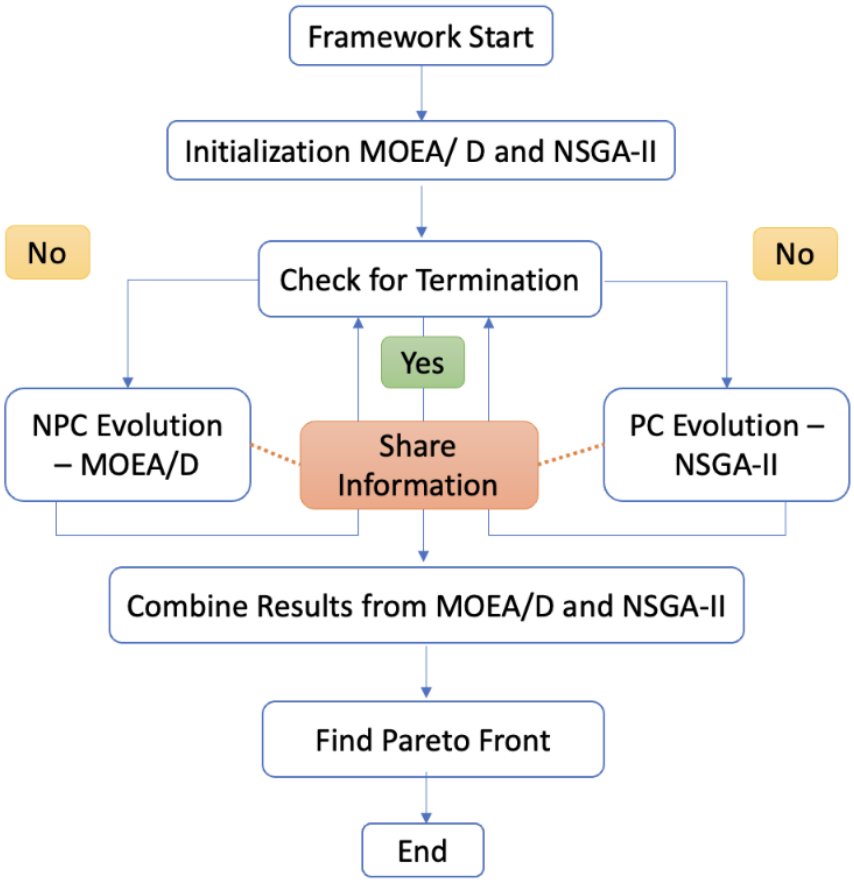

Figure 2.6: High Level Co^3MO Program Flow

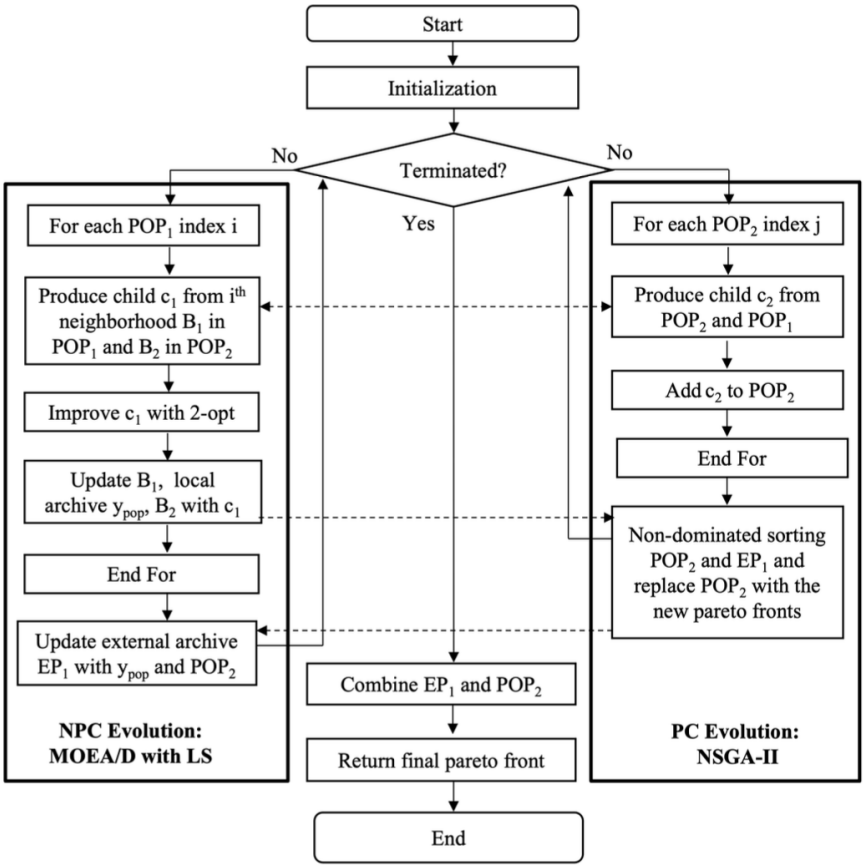

Figure 2.7: Co^3MO Program Steps [37]

There are four major steps in Co^3MO framework as shown in Figure 2.7 [37]:

- Initialization: NSGA-II population POP_2 and MOEA/D population POP_1 are both initialized randomly. n random weight vectors λ are generated. For MOEA/D, n decomposition cost matrices are generated; n neighbourhoods B_1 each of size T is also generated. To identify individuals for each neighbourhood population, pair-

wise Euclidean distances are calculated and compared between weight vectors. The single objective distance matrices (i.e., one for each decomposed scalar subproblem) are calculated based on the multi-objective distance matrix and λ using the weighted sum decomposition technique.

- Offspring Production: Different update mechanisms are used for each of PC and NPC evolution's:
  - For PC Evolution: For each individual in PC population POP_2, a child c_2 is created by using a recombination operator and is then added to PC population POP_2. In Co^3MO, a random number generator is used to select one of the hybrid operators (OX or IO) to generate the new child. OX stands for order crossover, which is an operator used for permutation problems. IO stands for inver-over, which is a combination of crossover and mutation. The mating pool for this recombination in PC evolution includes both PC population POP_2 (to preserve solution diversity) and MOEA/D population POP_1 (to further aid exploration and lead to convergence).
  - For NPC Evolution: MOEA/D uses a population set y_{POP} (a local archive which is an empty set initially) to temporarily

store the set of good solutions in this current generation. For each sub-problem, the mating pool includes both the MOEA/D neighbourhood B_1 and the NSGA-II neighbourhood B_2. Similar to the benefits gained by mixing both populations in mating pool as in PC evolution, this cooperation helps to diversify solution distribution and increase the exploration of entire search space. NSGA-II neighbourhood B_2 is calculated by finding T individuals from NSGA-II population POP_2 that are the closest ones to individual from MOEA/D population POP_1. The same hybrid operator used in PC evolution will be used to generate the new child c_1.

- Neighbourhood Update: For every NSGA-II neighbour in PC neighbourhood B_2, if the new child c_1 dominates it, replace it with the new child c_1. For every MOEA/D neighbour in NPC neighbourhood B_1, use the cost matrix to evaluate both that neighbour and the new child c_1; if the new child c_1 has a better fitness score, replace that neighbour with the new child c_1 and add the new child c_1 into the local archive population y_{POP} kept by NPC evolution.

- Population Maintenance: Recall from MOEA/D, there is an external archive population (EP) which holds the global best solu-

tions in all generation for MOEA/D. For the PC evolution, both NSGA-II population POP_2 and EP go through the non-dominated sorting and crowding distance selection. For the NPC evolution, EP is updated by using the current solutions stored in its local archive population y_{POP} and the solutions in NSGA-II population POP_2. If an old solution in EP is dominated by any of the new solutions, it will be removed from EP; and if a new solution is not dominated by any old solution in EP, it will be added to EP.

In [37], the algorithm is modified to adapt to changes in the Pareto front and Pareto set for solving MOP in dynamic problems. *In this thesis, we will consider static MOP only the static Co^3MO framework is used as the basis for parallelization.*

Chapter 3

Background on Parallel Computers

3.1 Graphic Processing Units

The graphic processing unit is an accelerator. It is designed for computationally intensive, data-parallel applications to achieve greater concurrency and performance. Data parallel computations are embarrassingly parallel computations that can be executed by each GPU code independently, concurrently, but synchronously. For example, if there are N cores, given two vectors, A and B of size N, the summation computation of each element i in the vectors ($A[i] + B[i]$, $i = 1, ..., N$) can be computed independently on N cores. This is because there is no dependencies between vector elements i and j in the summation operation. Each thread accesses it's own elements in A and B. The GPU provides provides large number of simple cores for exploiting embar-

rassingly parallel algorithms that exhibit no dependencies. As they are simple cores, there is operating system. Also, conditional statements cannot be executed in parallel. The GPU follows a single instruction multiple data (SIMD) model that exists in the early 1990's. Nvidia calls their GPU architecture with CUDA as single instruction multiple thread (SIMT) model.

As originally designed for accelerating graphic rendering applications, the modern GPU hardware is specialized in executing large number of threads simultaneously to achieve higher throughput. Modern GPUs provide greater flexibility with respect to the architecture and software for general purpose computations. Due to the large number of cores in the system, compute intensive computations can be off-loaded to the GPU and have become more attractive than CPU-based machines. The GPU has therefore, become extremely popular in executing machine learning algorithms that are compute intensive and easily parallelizable. Each layer in a deep learning neural network for example, can be executed in parallel. The computations are simple, repetitive and independent.

3.2 CUDA

CUDA is the programming software introduced by Nvidia in 2006 for designing general purpose applications on their GPUs [9]. A GPU executes *kernels*. Kernels are nothing but a function consisting of a series of instructions. A kernel is similar to a method in other standard programming languages. When a kernel is called, it will be executed by a number of different CUDA threads in parallel. And a thread refers to a basic unit of execution that performs a specific task on a GPU. There are two ways to launch a kernel function on GPU: (i) it can be called from the CPU, or (ii) it can be called from another kernel function on GPU.

In CUDA, kernel threads are organized into blocks, and threads of a block are executed in sets of 32 threads called *warps*. The current limit on number of threads in a block is 1024 [9]. Despite this limit, a kernel function can be executed by multiple thread blocks simultaneously, which gives the total number of used threads equals to the number of threads per block times number of blocks used [9]. Moreover, thread blocks are organized into grid [9]. Figure 3.1 [9] illustrate this structure.

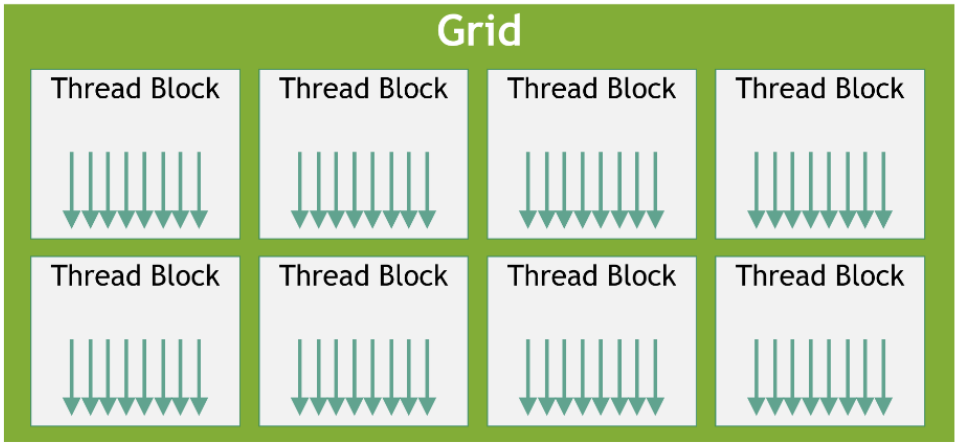

Figure 3.1: Thread, Thread blocks, and Grid in CUDA [9]

Further, thread blocks can also be grouped into clusters, and thread blocks in the same cluster are guaranteed to be co-scheduled just as threads in the same thread block do [9]. Figure 3.2 [9] illustrate this structure.

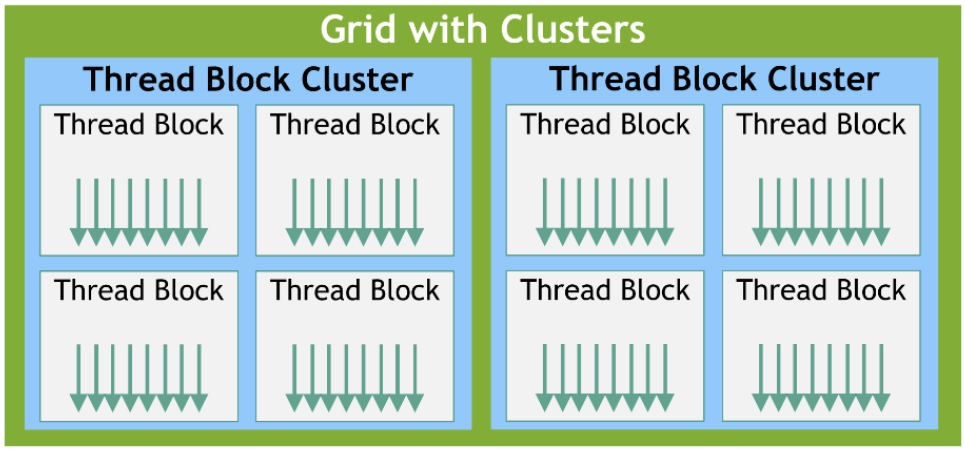

Figure 3.2: CUDA Grid with Cluster [9]

As Nvidia's GPUs follow a Single Instruction Multiple Thread (SIMT)

architecture [9], threads belonging to the same warp get to execute the same instruction at the same time. Programs with conditional statements (if and $else$ statements in C programming language) will not allow this. A thread in one warp may execute the if statement while another thread in another warp may execute the $else$ statement. This is not allowed in CUDA. In such circumstances, the conditional statements are executed sequentially. All threads will execute the if statement followed by the $else$ statement. This puts the threads at a disadvantage, decreasing performance. Therefore, it is imperative not to have conditional statements in the program when executed on a GPU.

CUDA also allows asynchronous operations. By allowing asynchronous operations in the context of SIMT, the CUDA program can continue its operations without waiting for the asynchronous task to complete. In other words, when an asynchronous task is initiated by a CUDA thread, it does not necessarily block the execution as that thread is not required to be included in the synchronizing threads [9].

In CUDA, each thread has its own private local memory. And each thread block has a shared memory that is accessible to all the threads of that block. Between thread blocks in a thread block cluster, read, write, and atomic operations can be performed on each other's shared

memory. Further, a global memory is accessible by all threads in a grid. Figure 3.3 [9] shows the memory hierarchy in CUDA.

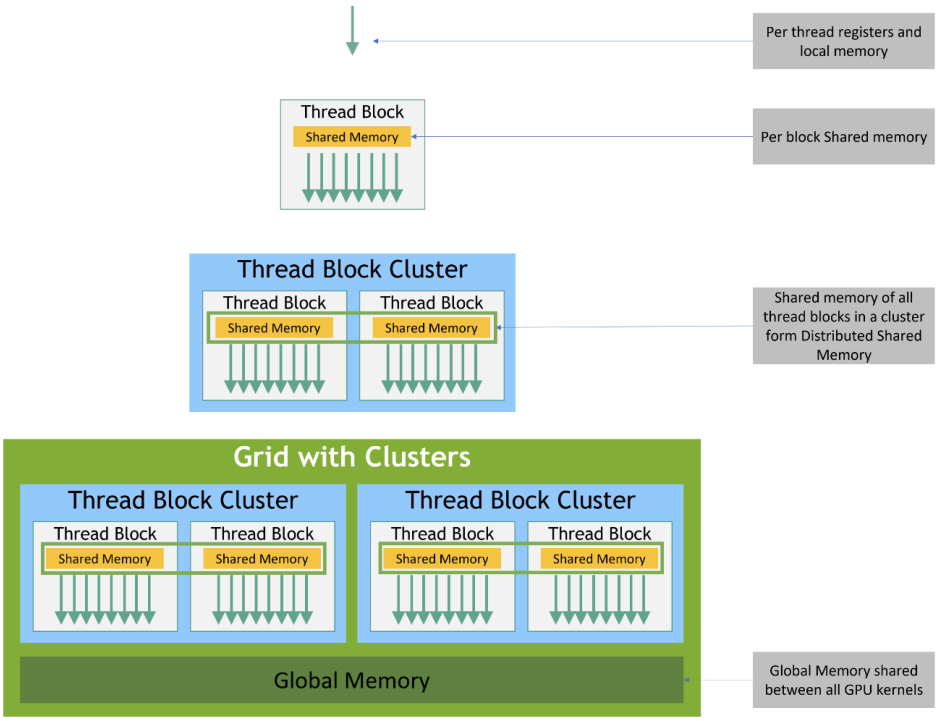

Figure 3.3: CUDA Memory [9]

To accelerate applications by running the computational intensive parts on GPUs, CUDA Toolkit developed by NVIDIA provides opportunities for researchers to develop, optimize and deploy their applications on GPU-accelerated embedded systems, HPC supercomputers, etc. GPU-accelerated libraries, debugging and optimization tools, a C/C++ compiler, and a runtime library to deploy the application are included in this toolkit [4]. Since C++ compiles the code for a program directly into machine code without intermediate translation at runtime,

programs written in C++ tend to have a faster execution time compared to the same programs in other interpreted programming languages like Python or Java. *In this thesis we will use CUDA Toolkit for C.++ to parallelize part of the NSGA-II algorithm in the BCE framework* Details are provided in Chapter 5.

3.3 CPU Multi-threading for Parallel Computing

CPU machines can be categorized as either distributed memory machines or shared memory machines. The difference between the two categories is how the processors communicate. Processors in distributed memory machines communicate by message passing; in shared memory, processors communicate through shared variables. The standard parallel programming languages used are Message Passing Interface (MPI) and OpenMP in distributed and shared memory machines, respectively. In shared memory machines, similar to a GPU architecture, threads can be created and executed on processors, providing multithreading.

CPU machines are more versatile than GPU machines. They follow a Single program Multiple Data (SPMD) model. There is no restrictions on the type of programs being executed on these machines. Programs can be executed asynchronously even with conditional statements. The

operating system runs on these machines providing a versatile and interface between the programmer, software and hardware supporting multithreading. In general, not all applications can be run on a GPU. The high complexity of developing non-data parallel algorithms on these machines over weighs the potential speedups that can be achieved. Algorithms that run on a GPU can be run on a CPU. The CPU is well optimized to run data-parallel algorithms. The main advantage of a GPU is that there is no operating interference and number of cores is large. But, if the algorithm is communication intensive then these machines are not suitable since accessing global memory is expensive degrading performance. And frequent transfers of data between CPUs and GPUs will also decrease performance.

MOEA/D is not completely data-parallel. The algorithm is more suitable for CPU based machines. *We use multithreading within a shared memory environment to design, develop, and implement a parallel MOEA/D algorithm.* A thread in a multithreaded architecture is responsible for each of the concurrent parts of a program, with its own path of execution [3]. Threads are lightweight processes within a process.

There are multiple benefits that can be gained from using C++

multi-threading for parallelizing a program:

- Significant improvement in terms of program overall performance, especially for compute and communication intensive applications.
- Potential for higher scalability for more powerful computing device.
- Better utilization of CPU usage.
- Complex computation can be further divided into smaller subparts, which can then be executed concurrently to improve performance and make each computation simpler.
- Can also be utilized in distributed computing.

Chapter 5 describes how we utilize this in our parallel MOEA/D model.

To further distinguish between the 2 types of CPU machines [1] [7] [8]:

1. Distributed Memory Machines - In general, since the members in a distributed memory system communicate by message passing, a high speed network is usually provided to interconnect the set of processing nodes. Each processing node consists of a processor and a local memory. If a given distributed memory machine has non-shared policy, each processor can only access its own local memory. To obtain information from other processors, messages

in the form of request/response will be sent over the network to the destined processors. In this way, data is moved and shared between the processors. Master-slave is a common paradigm when doing parallel computing with distributed memory machines. In this paradigm, a master computing node is in charge of distributing sub tasks to a set of worker nodes for fulfilling each sub task in parallel. The workers usually receive the same set of instructions but with different parts of the distributed data.

2. Shared Memory Machines - In contrast, shared memory machines allow processors to access the system memory directly without sending requests for permissions as they are built to connect to the same piece of memory. In other words, each processor has same access privilege to data that is created or used by any other processors. Despite the easiness in data access, shared memory architectures limits the memory capacity of a single machine and also the memory access speed as the processors are now needed to be put into a queue for data access.

Originally, the island model developed back in the year 1973 [30] was used to analyze the population differentiation in the study of the major racial groups of man and other species, where the number of groups

may be small and migration rates are low. Later, it has been adopted with modifications to study how to distribute genetic algorithms over multiple processors for solving optimization problems more efficiently in a parallel way [26]. Since each island can be responsible for a sub part of the problem, island model with multiple islands is well suited to be parallelized across a number of threads/processes in the CPU machines, as each thread/process can handle the computations for each island. *In our work, we parallelized islands' computation in thread-level since the costs of initiating new processes are much higher and the overhead is much more significant compared to the benefit.*

Chapter 4

Literature Review: Parallel EMO

As the thesis will hybridize NSGA-II and MOEA/D [37], two algorithms to drive PC and NPC evolution in the parallelized hybrid BCE framework, this chapter introduces existing works on parallel NSGA-II, MOEA/D and BCE frameworks.

4.1 Parallel NSGA-II

In the earlier years, there was lots of work done on genetic algorithm [2]. Genetic algorithms have been been studied for optimizing robot task scheduling problems [55], breast cancer diagnosis [13], resource scheduling on cloud [50], to name a few. Genetic algorithm has also been hybridized with other techniques [28, 51] techniques and for parameter tuning in machine learning [39, 46] models. The algorithm

is easily parallelizable [14, 24, 44]. The robustness and versatility of the algorithm, inspired the authors [18] to produce a software for solving optimization problems. In recent years, with the increased amount of computing power, and instinctive capability and effectiveness of NSGA-II, this algorithm has been considered for parallelization on many core architectures [25, 33]. There are works that have been done in the literature to develop parallel NSGA-II framework and utilize it to solve problems in MOP for various applications [21].

In general, there are two major compute intensive, time consuming parts in NSGA-II evolution process that is of most interest to be parallelized. The first part is the *offspring evaluation* process, where the objective functions for each individual is being calculated. The second part is the *non-dominated sorting*, where all the solutions are being sorted based on their dominance. Since the performance of both parts are largely dependant on the size of the population, effective parallelization means reduced overall computation time. The first part can be performed in parallel simply by distributing the fitness calculations across multiple computing units and collecting results back when finished, hence many efforts in the literature are focusing on parallelizing this part with different methods such as message passing based master-

slave model. The non-dominated sorting on the other hand is more complex when parallelizing compared to offspring evaluation since both the parent and child populations are needed when perform the sorting for classified fronts. In the literature, researchers solve this problem by partitioning the population and distributing them to multiple processors using either a fine-grained or coarse-grained approach.

In [19], three master-slave based approaches have been proposed for parallel NSGA-II, with the use of different synchronization mechanisms between the slave processors. In this work, the authors choose to parallelize the offspring evaluation step in the NSGA-II process, which is intuitive as the calculation for the cost of each individual in the offspring population can be computed independently and simultaneously. Among the three approaches, the synchronous version, which the authors referred to as the synchronous generational NSGA-II, allow the master process oversee the entire NSGA-II process. When the offspring need to be evaluated, the master process assigns each worker process an individual and let the worker calculate the value of the objective function for its assigned individual; after all the results have been sent back to master, a new offspring population is formed by selected individuals based on their evaluated fitness scores. In this approach, the number of

available worker processors is crucial to overall program performance. And since the size of the population is known apriori, there will be no extra benefit gained if the number of available worker processors is larger than the population size (as each worker processor will be responsible for one individual during the process). The other two proposed approaches follow an asynchronous pattern instead of the synchronous one. In one of them, all the available worker processors is used regardless of the population size. And for idled workers, new individuals are created by the master and sent to them for evaluation. This approach is being referred as asynchronous generational NSGA-II. Further, the master process need not wait for all the workers to return their results before moving on to the next generation; as in fact now the individuals generated in a later time can actually be inserted into the evolution process before the individuals that are generated earlier depend on the worker processors.

Asynchronous steady state NSGA-II is another approach based on asynchronous behaviour. In this approach, all available worker processors will be utilized in the same way as in previous approach; the difference is the use of steady state scheme in this version. The idea is that only one population is used in the entire evolution process, which

contains both parents and offspring. When new individuals are generated, they will be evaluated and compared to see if they are qualified to be incorporated in this population, if so, they will be included immediately.

Another work that tries to parallelize NSGA-II based on the classic Message Passing Interface (MPI) master-slave paradigm is by [52]. Similar to [19], the offspring evaluation step is the part that the authors parallelize in the NSGA-II evolution process. The main idea is to let the worker processors calculate the objective functions in parallel and let the master processor take control of the program flow and do the rest of the work. In this work, the goal is to solve multi-objective optimal power flow (OPF) problem by using the proposed MPI parallel NSGA-II.

In [10], two parallel NSGA-II implementations have been investigated for designing optimal water distribution networks. The authors referred to the first proposed model as the global model. Similar to [52], the master slave approach is adopted in this model where the objective functions are evaluated in parallel and computation time is therefore reduced. The second proposed model is based on the coarse-grained multiple population approach. The authors referred to it as the is-

land model, where the population is divided into a few sub populations called islands. The islands are evolved serially and independently with occasional migration between them where some of the solutions are exchanged. In this model, the search space is expected to be explored widely as the multiple sub population can evolve towards different directions. Convergence speed is also considered, as migration introduces diversity into each island and therefore helps in a faster evolution for each of them. In addition, the authors also claim better scalability for this model as the islands have low communication overhead between them. Despite such advantages, the island model for parallel NSGA-II needs to be tuned carefully for maximum performance in terms of solution quality and overall efficiency. The settings on island size, migration frequency, number of solutions for migration and migrating destined islands are crucial for this [10]. In this last model, the authors combine the global model with the island model in a way such that each island utilizes several worker processes to perform the solution evaluations simultaneously. In other words, each island now follows the master-slave model for its own evolution process, where the master process is responsible for collecting results from worker processes and communicating with other islands and the worker processes are responsible for

calculating the solution costs within each island.

As genetic algorithm is compute intensive very data parallel, GPU machines have been considered for parallelization. In [23], the non-dominated sorting and crowding distance selection NSGA-II have been parallelized on GPU. In non-dominated sorting, fronts are identified by launching a kernel on GPU, in an iterative manner. In this kernel, each individual is checked on its dominance against every other individual that has not been assigned a front yet. If the domination count for this individual is 0, then the front is assigned to this individual and its index is stored in a GPU array (which contains indexes of all individuals with domination count 0 for this current front). This array is used to update the front information at the end of each iteration by another kernel function, in other words one front is identified in each iteration. This process continues until all individuals have been assigned a front. For crowding distance selection, two arrays are defined on GPU, one for storing crowding distance for different individuals and another for storing the indices of each individual before the sorting (to keep the information on the original position of each individual). A kernel function is launched to update the crowding distance for an objective. This process is done for each objective in the MOP.

In [43], the authors proposed a GPU based parallel NSGA-II implementation to reduce the program execution time. In this work, multiple NSGA-II steps are executed on GPU but not all in parallel. The first parallelized step is the fitness evaluation, where each GPU thread is responsible for evaluating the objective functions for one individual. Again, this is intuitive to be performed in parallel as there is no need for communication between individuals in the evaluation process, since each thread only calculates the values of objective functions for the individual assigned to it. Another parallelized part is the non-dominated sorting step. In this work, the non-dominated sorting is divided into two stages, the first stage is to find the first front, and then the second stage is to find remaining fronts. The first stage is further divided into two parts. The first part determines the domination count for each individual and finds the individuals that are dominated by it. A kernel function is responsible for this procedure. The second part finds individuals that are not dominated by any other individual to become the first front. The second stage of non-dominated sorting in this work finds the remaining fronts by executing a kernel that computes one front at a time.

In [15], a parallel NSGA-II has been implemented on GPU for solv-

ing energy dispatch problems for hydroelectric power plants. Similar to previous works, objective function calculations, non-dominated sorting and crowding distance selection are the three parts that was parallelized.

In [34], fitness evaluation step of NSGA-II was implemented in parallel on GPU. *In our work, we parallelize the fitness evaluation step and non-dominated sorting step on the GPU.*

4.2 Parallel MOEA/D

A thread-based parallel MOEA/D was designed for multi-core processors in [41]. The idea is to distribute a portion of the population to a number of concurrent threads, and let each thread be responsible for the portion that is assigned to. Similar to the work proposed in [20] the authors partition the the whole population, with each partition composed of a number of different subproblems and is being evaluated and evolved in parallel. The difference from [41] lies in how the neighbourhood is defined for each subproblem. In [41], the neighbourhood is defined similar to the original MOEA/D algorithm, where the whole population is considered when defining neighbours for a subproblem. Therefore, the neighbourhoods can appear across different partitions.

Whereas in [20] the neighbourhood for each subproblem is defined by only considering the subproblems from the same partition. In [20], possible concurrent access to the same solution by different processors is also avoided compared to [41].

PaDe is another parallel algorithm based on MOEA/D and the island model [38]. The authors decompose a MOP into multiple subproblems based on MOEA/D. Each subproblem is assigned to an island with a predefined population size (i.e., the size of the island). And then each subproblem (island) is solved in a separate computational unit using the island model [49] to exchange good solutions between the subproblems (islands). In PaDe, islands evolve independently and simultaneously. Neighbouring islands are defined for each island. At certain intervals, the worst individuals of each island is replaced by the best individuals from each of its neighbouring islands. This exchange of information happens in an asynchronous manner, allowing island to receive immigrants at any time and send good solutions to its neighbours whenever it is ready. Finally the union of the best individuals from each island is returned as the final output population to the MOP. *In this thesis, we develop a parallel version of MOEA/D that is based on the island model but with synchronous communication between the islands.* Experiments

show that our version of parallel MOEA/D performs better in terms of both solution quality and execution time compared to sequential one.

Besides shared memory, work has also been done to parallelize a variation of MOEA/D with message passing clusters on distributed memory machines [53]. Another parallel MOEA/D implementation focuses on a virtual overlapping zone between partitions, and individuals are selected for mating and migration based on evaluation of individual populations in a certain area using the weight vectors of adjacent partitions [47]. MOEA/D has also been hybridized with other evolutionary algorithms, such as ant colony optimization, MOEA/D-ACO for solving MOPs [29]. The idea is to use any decomposition technique to divide a MOP into multiple single objective optimizing subproblems and also divide the ants into several ant groups for neighbouring purposes. Then let each ant work on one of the subproblems simultaneously. Since the communication is minimum between ant groups, it is also possible to parallelize as in [16], where the ACO part has been implemented using kernel functions on GPU.

Other related works include [35] where master-slave model and island model are considered for parallelizing MOEA/D for feature selection, and [54] where a GPU based parallel MOEA/D is proposed with

new task decomposition and scalarizaion methods.

4.3 Parallel BCE

In [22], a parallel hybrid algorithm is proposed for solving the complicated industrial chemical problem. The proposed algorithm is consists of two parts: the NSGA-II and the Successive Quadratic Programming (SQP). As one of the effective methods used for non-linearly constrained optimization problems, SQP approximates problem at each iteration with a local model that is represented by a simpler quadratic programming technique [11]. In this proposed framework, NSGA-II and SQP are executed independently, with periodic exchanges of information to improve their own results. Similar to the motivation behind Co^3MO by [37], this model allows the NSGA-II find evenly distributed Pareto points and use SQP to drive the convergence towards the real Pareto curve. By conducting a thorough literature review, we concluded there is lack of work in parallelizing the BCE framework. Therefore, in this thesis we will focus on the parallelization of BCE based on the work proposed by Liu et al. [37]. More precisely, we will implement a parallel version of NSGA-II on GPU and a parallel version of MOEA/D with multi-threading for a hybrid BCE framework.

Chapter 5

Hybrid Parallel BCE

In this chapter, we provide the design, implementation and evaluation of the parallelization of the hybrid sequential BCE framework [37]. The parallel BCE framework is composed of two processes, namely the PC evolution and the NPC evolution. Each evolution works independently to optimize its own solutions. At certain iterations, information in the form of solutions are exchanged between the two processes to help each other in the optimization progress. For NPC evolution, this would be to maintain and explore the search space more evenly and for PC evolution this would be to explore diverse points.

5.1 Parallel PC Evolution - NSGA-II

NSGA-II is chosen as the representative algorithm for driving PC evolution in our framework. Recall the major steps of NSGA-II can be summarized as follows (details are provided in Chapter 2):

1. Create Initial Population.
2. Breeding (Reproduction).
3. Combine Parent and Child Population.
4. Non-dominated Sorting.
5. Crowding Distance Selection.
6. Check for Termination.
7. Termination.

Figure 5.1 provides a visual walk through for the algorithm flow.

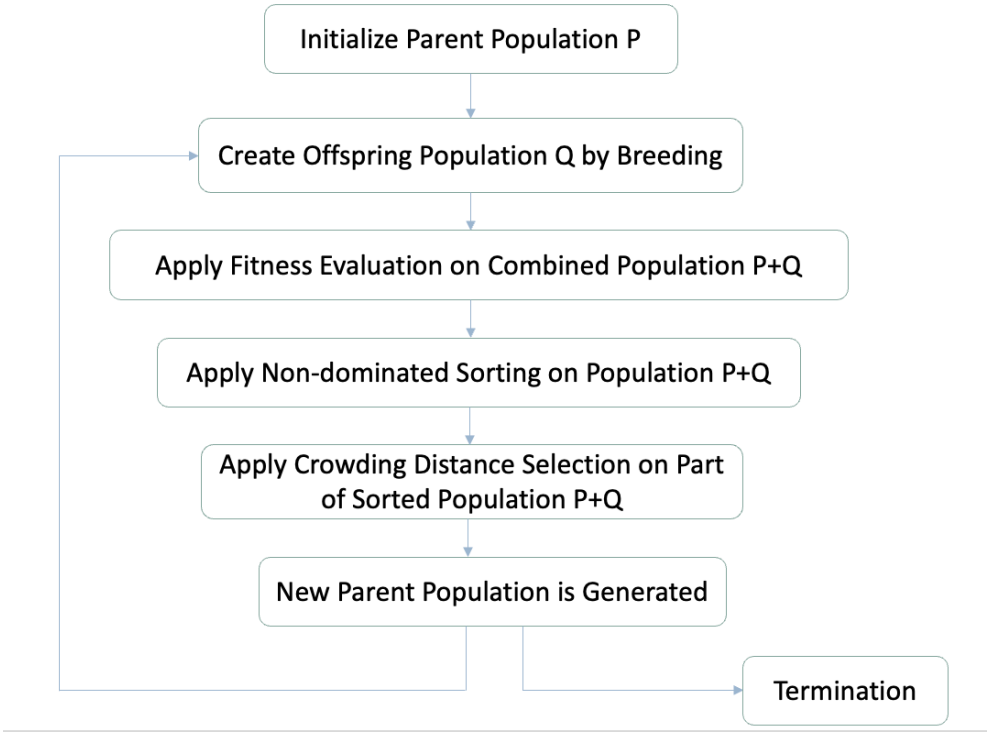

Figure 5.1: NSGA-II Algorithm Flow Chart

By studying the sequential algorithm, performing experiments, and reading the literature, we found that two of the above steps are very computationally intensive and dominate the overall execution time in the NSGA-II algorithm. The two steps are fitness evaluation and non-dominated sorting:

- Fitness Evaluation: After combining the parent and child population to form a new combined population, the fitness is evaluated on each individual in this new combined population. Let m denote the number of objectives in a given MOP, then the fitness evalu-

ation calculates the value from each of the m objective functions, for each individual in the combined population. Using Figure 5.1 as an example, costs of the m objective functions will be calculated for every individual in the combined population $P+Q$.

- Non-dominated Sorting: Individuals in the combined population will then be sorted according to their level of dominance compared to other solutions in the population. Let F denote the front. Then $F = (F_1, F_2, \ldots)$ are all non-dominated fronts after sorting is applied.

The values of the objective functions for each of the individuals can be calculated independently. Therefore, this is easily parallelizable. The fitness procedure can be done in parallel. Non-dominated sorting on the other hand involves comparisons of fitness scores between each individual and other individuals to identify each individual's front. These comparisons can be done in parallel and therefore the non-dominated sorting procedure can be parallelized. Figure 5.2 highlights the two function that are being parallelized in this work.

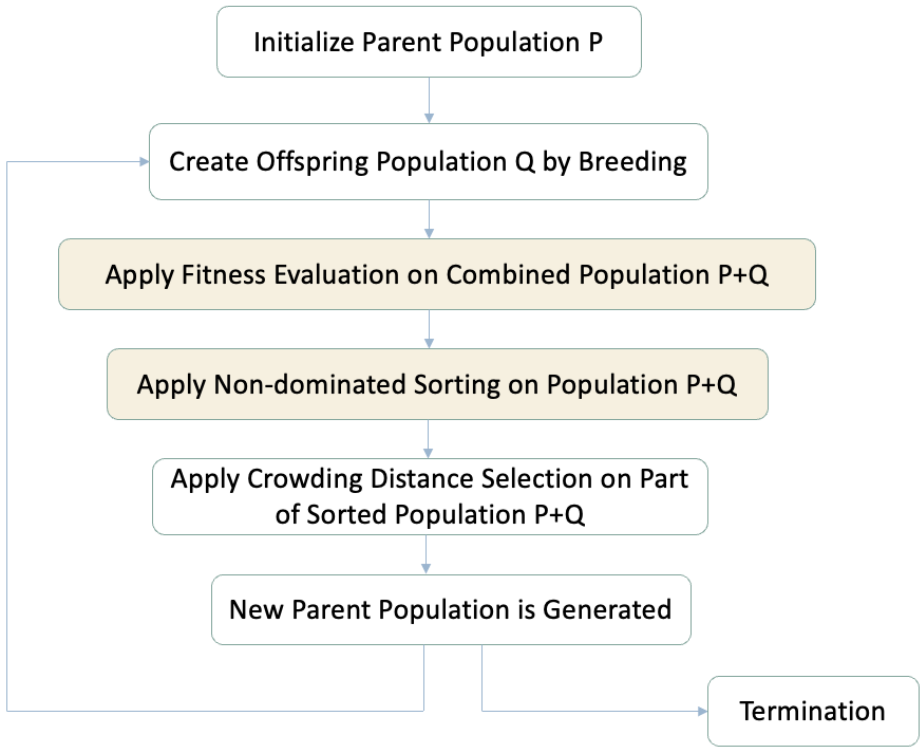

Figure 5.2: The Two Parallel Parts in Our NSGA-II Implementation

The following subsections 5.1.1 and 5.1.2 give details on how we parallelize each of the function on GPU using CUDA.

5.1.1 Parallel Fitness Evaluation

In using the GPU machine, it is important that we keep the cores busy by executing the threads on these cores. We assign GPU thread for calculating the costs of m objective functions for one single individual solution in the population. For example as shown in Figure 5.1, we will use $P+Q$ GPU threads for a population of size $P+Q$. All data

needed for calculating the fitness scores are first transferred from the host machine, CPU, to the device, GPU global memory. The GPU has a separate global that is accessible by all the threads for faster access. Data transfer is always the first step in working with GPU. All kernels (programming functions to be executed on the GPU) is called from the CPU. There are 4 steps in computing the parallel fitness evaluation:

1. Data Transfer: Data needed for calculating the fitness scores are transferred from CPU to GPU, and are stored in GPU's global memory for threads to access. More precisely, the following information is sent from CPU to GPU:

   - population $P+Q$
   - m cost matrices (in this work, we focus on solving bi-objective TSP used in the benchmark dataset, so $m = 2$ in this case)
   - graph rank
   - population size $P+Q$
   - an empty output matrix (for allowing the GPU to fill in the calculated costs)

2. Kernel function: A kernel function for calculating the fitness scores defined on the CPU. The kernel is defined using $\_global\_$ declaration

specifier and the number of CUDA threads (N) for a given kernel call is specified using a new <<< ... >>> . The kernel is initiated by the CPU call and is executed on the GPU.

3. Fitness score computations: The actual code for the computations of the fitness score is written in the kernel function mentioned in point 2 above. Fitness scores for all individuals are calculated in parallel on the GPU. That is, the kernel is called by each of the N threads specified in the call and executed by the streaming multiprocessors on the GPU.The empty output matrix are filled in with the costs.

4. Data transfer: GPU sends the results back to the CPU. That is, the filled in output matrix is sent back to CPU.

Pseudo code for describing the parallel fitness evaluation can be found in figure 5.3.

```
# each thread will calculate the scores for one tour (objective 1 and objective 2)
note: block_size =ceil( len(population) / MAX_THREAD_PER_BLOCK )

CPU: cuda_motsp_score[block_size, min(len(population), MAX_THREAD_PER_BLOCK)](population,
cost_matrix1, cost_matrix2, rank, len(population), output);

# rank is the length of a tour
GPU: cuda_motsp_score ( population, cost_matrix1, cost_matrix2, rank, len(population), output):
    threadId, cost1, cost2
    for i … rank - 1:
        point1 = population[threadId][i]
        point2 = population[threadId][i+1]
        cost1 += cost_matrix1[point1][point2]
        cost2 += cost_matrix2[point1][point2]
    output[threadId][0]  = cost1
    output[threadId][1] = cost2
```

Figure 5.3: Pseudo Code for Parallel Fitness Evaluation

5.1.2 Parallel Non-dominated Sorting

For non-dominated sorting, the goal is to classify all the individuals in the combined population into different fronts based on the level of dominance of each individual. After applying non-dominated sorting, crowding distance selection will also be applied to the last partially filled front (if there is one) to further select individuals in that front to be included in the new population. One thing to note is that the size of the population will be reduced after selection. For example, in Figure 5.1, let N denote the size of P. We have a combined population $P+Q$ with size $N+N$ before applying non-dominated sorting. After applying

the sorting and selection, the new population we have is of size N. And this new population will either be the new parent population for generating a new child population in the next iteration of the program or be the final population as the output to the program, depending on the terminating condition (Section 2.2).

As explained earlier, the number of individuals that require crowding distance selection is relatively small. After performing experiments on the crowding distance selection procedure, we realized that initiating a kernel call for the this function on the GPU is slow. As a result, crowding distance selection procedure is not parallelized.

The implementations details for non-dominated sorting is as follows. We let each GPU thread be responsible for checking if one tour is a non-dominated solution for the current rank of the Pareto front. All ranks of Pareto fronts (i.e., F_1, F_2, ..., F_i, ..., F_n) will be identified in an iterative manner. More specifically, this is performed within a loop. The loop is executed on the CPU. Every front F_i is filled with individuals until the size of the set of Pareto fronts is equal to or larger than the size of the initial population P (i.e., if the size of current Pareto fronts is less than the size of P). The iterative procedure within a loop consists of the following:

- Preparation of the data to be sent to the GPU.
- Call the CUDA kernel function on GPU for identifying individuals who are non-dominant from the current unselected individuals in this current front F_i.
- Send the results back to the CPU. There is synchronization between the time the kernel is called and the time the results are received by the CPU.
- CPU adds the newly identified members for this front into the Pareto fronts.
- Check on the size of the current Pareto fronts is performed. If the size exceeds the size of initial population (i.e., size of P, which is N), crowding distance selection will be applied to select individuals from the last front F_n to be included in the new population of size N. Otherwise the loop continues to a new iteration if the condition met.

The implementation for each *i-th* iteration of the loop is given below in more detail:

1. Data Transfer: Data needed for sorting the individuals for the current front are sent and stored in GPU's global memory. These

include:

- fitness scores of unselected individuals up to this current iteration in the population $P+Q$. In other words, these individuals are not being included in any front yet.
- an output array filled with 1's. This array is used by the GPU to mark the dominated individuals for this current iteration/front.

2. CPU calls GPU kernel function: This is the non-dominated sorting function. It is called using the $\_global\_$ declaration specifier and the of number of CUDA threads.

3. The function is executed by each of the threads and executed on the streaming multiprocessor. The threads checks whether each unselected individual is non-dominant or not, compared to other unselected individuals. Front F_i is identified by marking individuals that are not non-dominant with 0's. In other words, the original array received from CPU (filled with 1's) is now being modified to include 0's for representing individuals that are not being selected for this current front (i.e., F_i).

4. Data Transfer from GPU to CPU: Results are sent back to CPU:

The updated array filled with 1's (representing selected individuals for this front F_i) and 0's (representing unselected individuals for this front F_i) is sent back to CPU.

5. On CPU, one process then checks if the size of F_i + the size of *all previous fronts* F_1 *to* F_{i-1} is less than the size of P; if so, it means all individuals in front F_i could be included into Pareto fronts without applying crowding distance selection. In this case, we will add F_i to the final set of Pareto fronts, and remove individuals in F_i from the unselected population (which will be used as input in the next iteration). Otherwise, if the size of F_i + the size of *all previous fronts* F_1 *to* F_{i-1} is larger than the size of P, then we have too many individuals in the last front F_i. Therefore, crowding distance selection (on CPU) is applied on the last front F_i to select a portion of individuals to be included in the final Pareto fronts set.

Figure 5.4 provides pseudo code for our parallel non-dominated sorting implementation.

```
# each thread will check on if one tour is the non-dominated solution for the current pareto front
CPU:
unselected_population_ids = all population ids
pareto_front = []
while (len(pareto_front)) < initial population size:
        output = array with len(unselected_population_ids) filled with 1s
        scores = scores for all the unselected population ids
        #  call cuda kernel function to check on each tour to see if it is a non-dominated solution
        cuda_identify_pareto_front[1, len(unselected_population_ids)](scores, output)
        current_pareto = output
        remove current pareto from unselected population ids
        add current pareto to pareto_front
        if (pareto_front size > init population size):
                use crowding distance selection to find remaining solutions in last pareto front Fn

# scores: scores for unselected populations
# output: an array of 0s and 1s -> 0 means a solution is not non-dominated, 1 means a solution is in the pareto front,
GPU: cuda_identify_pareto_front(scores, output)
        threadId, size = len(scores)
        #compare tour [threadId] with all other tours to see if it is non-dominated solution
        for i in 1… size:
                if threadId != i
                        scores1 = scores[threadId]
                        scores2 = scores[i]
                        # check if scores 2 wins
                        if (scores2[0] <= scores1[0] and scores2[1] <= scores1[1]) and (scores2[0] < scores1[0] or scores2[1] < scores1[1])
                                # tour i dominates tour [threadId], mark tour [threadId] as dominated (0)
                                output[threadId] = 0
                                break
```

Figure 5.4: Pseudo Code for Parallel Non-dominated Sorting

5.2 Parallel NPC Evolution - MOEA/D

For NPC evolution, we will use a variation of the original MOEA/D algorithm. Recall the major steps of MOEA/D are the following (details are provided in Chapter 2):

1. Initialization and Decomposition
2. Reproduction
3. Improvement (optional).

4. Update Neighbourhood

5. Update External Population (EP)

6. Check for Termination

Also recall from Chapter 4, MOEA/D was proposed by [38] to be used with an island model for parallel execution with asynchronous communication between the islands. In our work, we will use a similar model based on the island prototype for our parallel MOEA/D implementation. The major difference between islands in our model and islands in [38] is that our islands follow a synchronous communication pattern instead of asynchronous information exchanged. This ensures that every island evolves at the same pace, and information exchanged between an island and its neighbours are always up-to-date. Otherwise exchange of stale information has a risk of leading the optimizing direction of an island towards sub optimal regions and is therefore a waste of overall efficiency. Figure 5.5 illustrates the idea of our parallel MOEA/D implementation with the incorporation of synchronous island model.

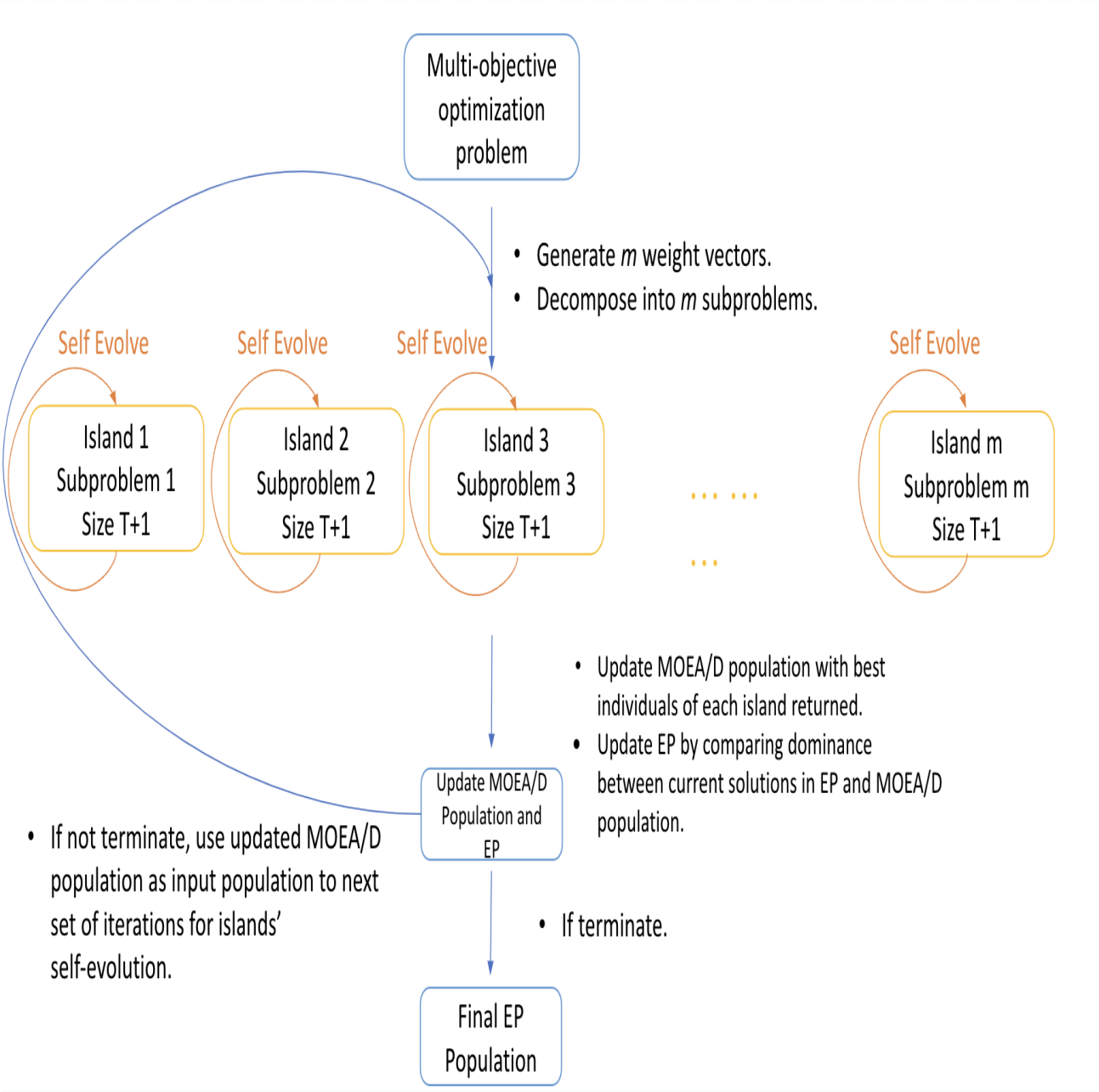

Figure 5.5: Our Parallel MOEA/D with Island Model

The following explains our model with respect to Figure 5.5:

- Initialization and Decomposition: To begin with, let m denote the number of weight vectors which is also the number of subproblems.

To find m, a control parameter H will be used as in the original MOEA/D. Equation 5.1 shows how to find the value of m based on H, as each weight component takes a value from $\frac{0}{H}$, $\frac{1}{H}$, ..., $\frac{H}{H}$ and $\lambda^1, ..., \lambda^m$ are all the m weight vectors.

$$m = \binom{num(objective) - 1}{H + num(objective) - 1} \tag{5.1}$$

Since we are focusing on the bi-objective TSPs, the dimension of m will be 2, which means for each scalar optimization subproblem, we have 2 weighted components, one for each objective in the objective function. Then by using the weighted sum approach, we decompose the bi-objective TSP into m single objective subproblems.

- Assign Subproblem to Island: Each subproblem will be assigned to a single island for finding optimal solutions specific to that subproblem. In other words, each island will have its own evolution process towards its best solutions. In this way, islands are well-suited for evolving in parallel. For each island, a neighbourhood of islands of size T is defined. The value of T is chosen to be a smaller number compared to m. And the size of population on each island is set to be *T+1*, since in the later stage the worst T solu-

tions in an island will be replaced by the good solutions from its T neighbouring islands. Similar to the concept of neighbourhood in MOEA/D, the neighbours of an island are assigned based on the closest pair-wise Euclidean distances between weight vectors of itself and each of the other islands.

- Islands Self Evolving: Each island then undergoes the following steps for a predefined number of iterations (user-defined):
  - Reproduction: Similar to generic breeding method, two individuals in the population (recall, this population contains the island itself and the current best solutions from each of its neighbours) within each island are selected for undergoing genetic operations.
  - Improvement (optional): Not considered in this work. And from experiments, despite not having any improvement techniques such as local search, the results returned by our parallel MOEA/D model are still comparably good in terms of both execution time and solution quality.
  - Update Neighbourhood: Once a new child is produced, it is compared to the worst solution in the island's current neighbourhood. If the new child has better score, the worst solution

in the island's neighbourhood will be replaced by this new child (i.e., the neighbourhood is updated); else the new child will not be stored (i.e., the neighbourhood will not be updated).

- Islands Communication: Communication in our implementation is performed in the form of population updating instead of exchanging solutions directly between the neighbouring islands. With synchronous updating, the MOEA/D population will be updated once all islands complete their self-evolution for this current iteration, this is marked as *Update MOEA/D Population* step in figure 5.5. During the *Update MOEA/D Population* step, each island sends its current best solution to the MOEA/D population. The updated MOEA/D population is used as input population in the next round of iterations for islands' self-evolution, otherwise terminate.

- Update External Population (EP): Update EP by comparing current solutions in EP to the solutions in updated MOEA/D population based on dominance.

- Terminate: If terminating condition is met, output final EP as final result.

Figure 5.6 describes the procedures for our parallel MOEA/D with

islands incorporation in the form of pseudo code.

```
define_neighbourhood();  # for each island (i.e., each subproblem), find its T closest neighbours based on pairwise Euclidean distances of the subproblems' weight vectors

# num_iterations is a predefined constant, represents total number of iterations for islands evolution

for iteration in num_iterations:

    generate_islands;  # generate islands for all subproblems, each island is responsible for a subproblem and each island stores information of its neighbourhood for convenient access without interfering each other's evolving process

    # assign each island to a worker thread

    for island in islands:
        thread worker (worker_function, island)  # assign an island to a worker thread and invoke worker function. All worker functions will be executed in parallel by worker threads.

    y_prime_pop;  # a temporary set to store current best solutions from all islands  in each iteration; reset for each iteration

    # find the best result from each island, update MOEA/D population

    for island in islands:

        best_result = find the best solution in the island's updated neighbourhood (including itself)

        y_prime_pop.add(best_result)

        population[index of island] = best_result;  # update MOEA/D population with the best tour

    update_external_population(y_prime_pop)


#  below is the thread function

worker_function (island):

    for iteration in (1, S):  # let S be the number of iterations for each island's self evolution

        child  = reproduction (island)  # breeding by using genetic operation

        if child has a better score, replace the worst solution in the neighbourhood (including itself) with the child; else discard child

    return the updated island;
```

Figure 5.6: Pseudo Code for Parallel MOEA/D with Island Model

5.3 Parallel BCE - PC and NPC Communication

Recall from Chapter 2, the goal of any BCE framework is to take advantage of both PC and NPC evolution. To do so, sharing of useful information is needed between the two evolution algorithms. Based on [37], for each algorithm, the following describes how it utilizes the other algorithm's information to facilitate its own optimization process in our implementation:

1. Framework start
2. Initialize parallel NSGA-II and parallel MOEA/D.
3. If not terminate, NSGA-II and MOEA/D will run in a sequential order. When running, each of them will perform parallel computations separately. This step is executed iteratively until terminating condition is met.
    - For NSGA-II evolution process, good solutions from MOEA/D is used in helping the search direction to not miss the extreme points:
        - MOEA/D population is used when producing new offspring in the offspring reproduction step.

  - MOEA/D external population is used together with current NSGA-II population in the non-dominated sorting step to determine new Pareto fronts and update NSGA-II population based on this.

- For MOEA/D evolution process, it uses NSGA-II's good solutions to avoid losing the even distribution for the population:
  - NSGA-II neighbourhood from NSGA-II population is used when producing new offspring in the offspring reproduction step. To clarify, NSGA-II neighbourhood is defined for MOEA/D computation, and is selected by closest pairwise Euclidean distances on the fitness scores between each MOEA/D island to the solutions in NSGA-II population.
  - NSGA-II population is used together with current best solutions of MOEA/D when updating the external population of MOEA/D.

4. If terminate, combine NSGA-II population and MOEA/D external population. Compute and return Pareto front of this new set as final result.

By combining our parallel NSGA-II and parallel MOEA/D evolutions, pseudo code in figure 5.7 shows the overall program procedures of our

parallel BCE framework.

```
for gen in generations:

    # execute parallel NSGA-II for PC evolution

    if (gen == 1)
        nsgaii_parent_population = nsgaii.population + moead.population  # no external population from MOEA/D yet for the 1st iteration
    else
        nsgaii_parent_population = nsgaii.population + moead.external_population  # MOEA/D sends information to NSGA-II
    nsgaii.breed_population(nsgaii_parent_population)

    #  find pareto front and set them as new nsgaii population
    nsgaii.population = find_pareto_front_cuda and crowding_distance_selection;

    #  execute parallel MOEA/D for NPC evolution
    define islands;  # array to store all islands

    for each subproblem in MOEA/D.population:
        generate island and island's neighbourhood;
        nsgaii_neighbours = find neighbour in nsgaii(tour, nsgaii scores);  # finds neighbours in NSGA-II population based on shortest
Euclidean distance of current MOEA/D subproblem to each solution in NSGA-II population
        island.neighbours.append(nsgaii_neighbours);  # MOEA/D stores information from NSGA-II in the form of neighbours
        islands.append (island)

    for island in islands:
        self evolve in parallel;

    moead.update_population();  # update MOEA/D population with updated islands
    moead.update_external_population();  # update external population

#  for final result of BCE
final_output = union (moead.external population, nsgaii.population);
```

Figure 5.7: Pseudo Code for Parallel BCE

In the next chapter, the parallel algorithm experimental results are compared to sequential results using benchmark datasets, efficiency,

computation and communication overheads, etc. are discussed.

Chapter 6

Experiments and Results

Recall the main benefit gained from parallelizing any sequential algorithm is to better utilize available resources and therefore achieve a faster program execution time. In other words, parallel implementation should give us better efficiency in terms of program execution time without sacrificing solution quality. In this thesis, we have checked the quality of final solutions produced by our parallel framework through metrics such as Inverted Generational Distance (IGD) scores, and it is compared to the final results provided in the sequential implementation from [37]. By setting the configuration of all variables and parameters to be the same as the sequential algorithm in [37] and conducting set of experiments on various data sets, we have confirmed our parallel framework has achieved better performance in terms of both execution time and solution quality.

In terms of better solution quality, we find that our modified version of MOEA/D with island model contributes the most to the significant improvement in IGD scores as it drives the BCE framework towards better fronts, as shown in Figure 6.14. All datasets and best known results used in this thesis are obtained from Biobjective TSP online open resource [6]. We performed our experiments on 3 of these benchmark datasets: euclidAB100, euclidAB300 and euclidAB500 for data size of 100, 300 and 500. Each file provides the Euclidean coordinates for each node.

For genetic operator used in offspring production (i.e., Reproduction) step, we choose to use cross mutation for all our experiments. We have applied several genetic operators including ordered crossover, inver over, and cross mutation. Among them we found cross mutation produced final results with best IGD scores in our experiments. Therefore, this operator is chosen in this thesis.

NVIDIA GeForce RTX 3090 is used for running all GPU related tasks in this work. CPU processor used is Intel i5-10400F. All programs for this thesis are written in C++. The sequential implementation in Python is obtained from our IDEAS Laboratory, PhD thesis by Ying Ying Liu [36] at the University of Manitoba. Note the differences be-

tween C++ and Python in terms of performance is not included in the analysis for the scope of our work. Details of experiments and visualization of results can be found below.

6.1 Execution Time

To better evaluate the speedup of our parallelization, we first analyze and evaluate the execution time for each component (NSGA-II and MOEA/D) separately, and then evaluate the overall framework.

6.1.1 Parallel NSGA-II Speedup

First, we look at the speedup for our parallel NSGA-II implementation. This has been further divided into 3 parts: 1. the parallel fitness evaluation part, 2. the non-dominated sorting part, 3. the overall NSGA-II execution time. In this work, we use the definition from [5] to define speedup. According to [5], speedup is a measure that captures the relative benefit of solving a problem in parallel compared to the sequential version. More specifically, in our work it is expressed as the ratio of the execution time of the sequential implementation to the execution time of the parallel implementation.

Parallel Fitness Evaluation Speedup

For our parallel fitness evaluation on GPU, we performed experiments on euclidAB100, euclidAB300 and euclidAB500, with various size of BCE population. The number of iterations is set to 50. Figure 6.1 shows the execution time for both the parallel and the sequential fitness evaluation function on different runs of experiments. As from the Speedup column, we can see the benefits gained by using GPUs is especially obvious when the size of the population increases, and this advantage is even more significant when the dataset is large. From this observation, we verified that GPU is very suitable for parallelizing compute intensive applications, and this is especially true for larger size dataset as the GPU resources are better being utilized. On the other hand, for small dataset, not performance is gained and the GPU architecture may not be beneficial. The GPU resources are not well utilized and therefore are wasted. The data transfer time between the CPU and GPU may outweigh the overall performance for smaller dataset. The data transfer is the most costly operation in using the accelerator and although the transfer time cannot be reduced, it would be tolerated if there are other kernels that can be executed in parallel on the CPU. In our algorithm, this is not the case.

CROSS_MUTATION	Execution Time for Parallel NSGA-II Fitness Evaluation				
Dataset	Population Size	# of Iterations	Average Time (in milliseconds) - GPU	Average Time (in milliseconds) - CPU	Speedup
	50	50	32.234	173.115	537.06%
	100	50	46.838	364.033	777.22%
	200	50	62.977	764.012	1213.16%
	500	50	85.156	1930.02	2266.45%
euclidAB100	1000	50	136.436	3715.993	2723.62%
	50	50	58.436	583.012	997.69%
	100	50	62.438	1136	1819.40%
	200	50	114.559	2314.094	2020.00%
	500	50	309.199	6421.166	2076.71%
euclidAB300	1000	50	328.139	11896.87	3625.56%
	50	50	65.034	1104.097	1697.72%
	100	50	117.425	2183	1859.06%
	200	50	241.208	4364.07	1809.26%
	500	50	568.944	12145.159	2134.68%
euclidAB500	1000	50	668.013	23105.054	3458.77%

Figure 6.1: Comparison of Execution Time for Parallel NSGA-II Fitness Evaluation and Sequential NSGA-II Fitness Evaluation

Further, Figure 6.2 shows the execution time for different population sizes for the 3 data sets on GPU. As expected, the execution time is increases for larger population size. However, we can see from Population size = 500 and beyond, the execution time stops growing linearly, unlike for Population size less than 500. For larger population sizes, the number of blocks (threads) increases. The blocks are distributed in terms of warps by the scheduler and the resources of the GPU are well utilized. More than two warps maybe assigned to a GPU core. As the threads are performing the same computations concurrently, the efficiency of the GPU increases.

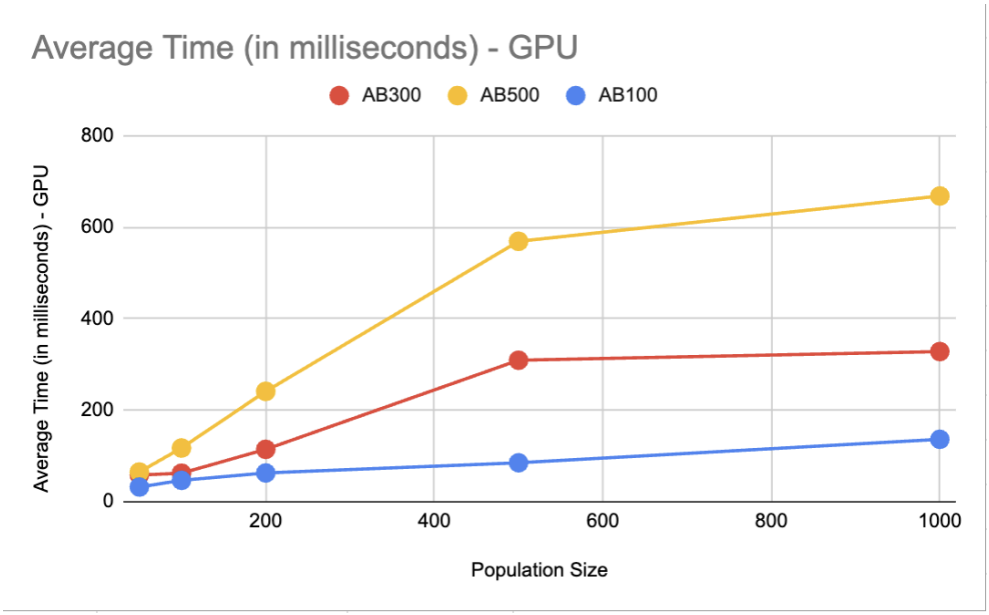

Figure 6.2: Parallel NSGA-II Fitness Evaluation - Execution Time for Various Population Size

Parallel Non-dominated Sorting Speedup

Next, for our parallel non-dominated sorting on GPU, we performed similar experiments on euclidAB100, euclidAB300 and euclidAB500, with various sizes of BCE population. The number of iterations is again set to 50. Figure 6.3 shows the execution time for both the parallel and the sequential non-dominated sorting function on different runs of experiments. And Figure 6.4 visualizes the execution time for different population sizes for the 3 data sets on GPU. The average execution time for non-dominated sorting is longer than the time for fitness evaluation as a result of more complex computations (i.e., comparisons

and more conditional branching statements instead of simple calculations) required to mark the dominance of individual solutions. Note, that the accelerator performance is greatest when the computations are data parallel. When a thread encounters a conditional statement, the threads execute the instructions sequentially. Also, the conditional statements provide options to the threads as to which statements to execute ("if" or "else"). In this case, some threads executing the "if" maybe active, while other threads idle and vice versa. Therefore, it is important that conditional branch statements are not executed on the GPU as these machines do not have a branch predictor like the CPU. For the algorithm under consideration the resulting speedup is still considerably better given that the CPU computation is very slow for large population sizes.

CROSS_MUTATION	Execution Time for Parallel NSGA-II - Non-dominated Sorting				
Dataset	Population Size	# of Iterations	Average Time (in milliseconds) - GPU	Average Time (in milliseconds) - CPU	Speedup
	50	50	40.243	677.985	1684.73%
	100	50	67.103	2599.922	3874.52%
	200	50	127.287	10014.041	7867.29%
	500	50	506.455	63202.107	12479.31%
euclidAB100	1000	50	2764.881	252944.078	9148.46%
	50	50	49.185	673.051	1368.41%
	100	50	77.868	2711.9	3482.69%
	200	50	203.974	10007.058	4906.05%
	500	50	2099.598	65713.896	3129.83%
euclidAB300	1000	50	4619.517	264245.965	5720.21%
	50	50	52.649	743.952	1413.04%
	100	50	108.817	2720.9	2500.44%
	200	50	421.015	10370.058	2463.11%
	500	50	2677.888	66278.638	2475.03%
euclidAB500	1000	50	7767.711	247850.01	3190.77%

Figure 6.3: Comparison of Execution Time for Parallel NSGA-II Non-dominated Sorting and Sequential NSGA-II Non-dominated Sorting

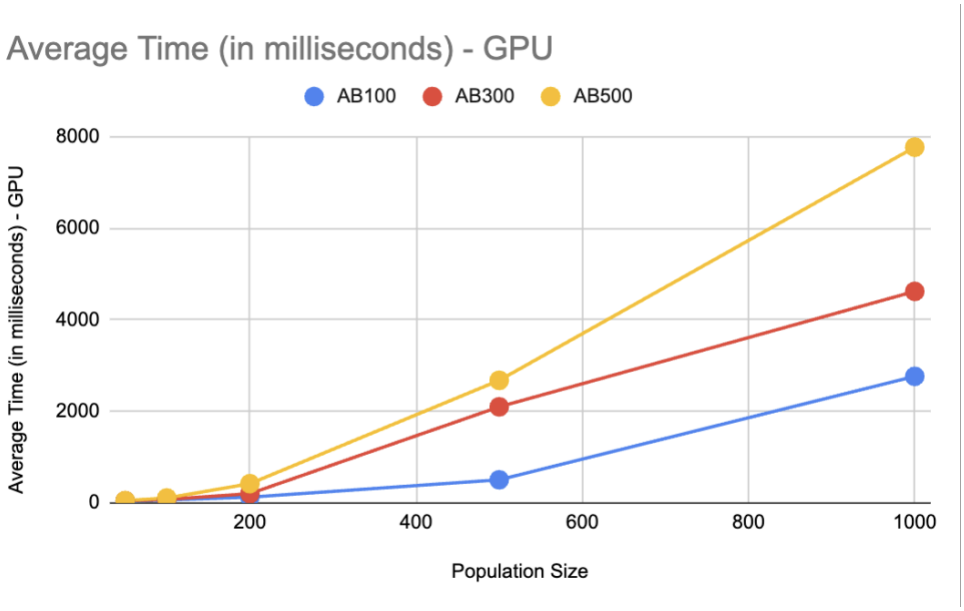

Figure 6.4: Parallel NSGA-II Non-dominated Sorting - Execution Time for Various Population Size

Parallel NSGA-II Overall Speedup

Lastly, we look at the overall execution time for parallel NSGA-II with above two parts being parallelized on GPU. Figure 6.5 provides specifications on speedup and Figure 6.6 visualizes the growth curve of the execution time for increased population size.

CROSS_MUTATION	Execution Time for Parallel NSGA-II - Overall				
Dataset	Population Size	# of Iterations	Average Time (in milliseconds) - GPU	Average Time (in milliseconds) - CPU	Speedup
	50	50	633	6268.9	990.35%
	100	50	1038	13823	1331.70%
	200	50	1931	32491	1682.60%
	500	50	5407	122150	2259.11%
euclidAB100	1000	50	14944	366680	2453.69%
	50	50	2502	71316.01	2850.36%
	100	50	4670	141381	3027.43%
	200	50	9400	301405	3206.44%
	500	50	26442	825696	3122.67%
euclidAB300	1000	50	56875	1801549	3167.56%
	50	50	5796	274869.071	4742.39%
	100	50	11699	555893	4751.63%
	200	50	23255	1152077	4954.10%
	500	50	62625	3012328	4810.10%
euclidAB500	1000	50	138145	6248727	4523.31%

Figure 6.5: Comparison of Overall Execution Time for Parallel NSGA-II and Sequential NSGA-II

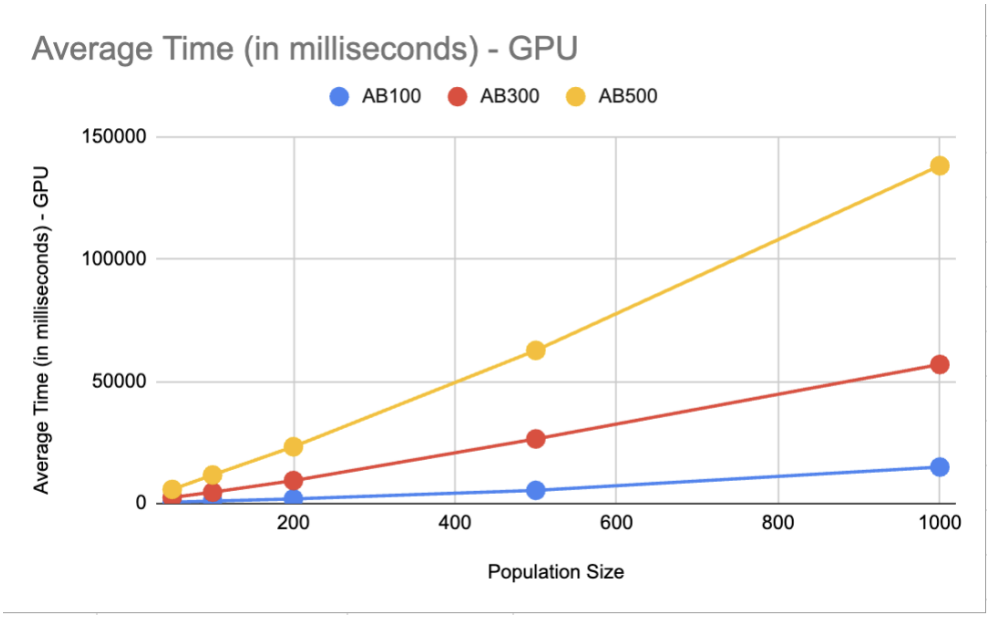

Figure 6.6: Parallel NSGA-II - Overall Execution Time for Various Population Size

The growth curve for overall NSGA-II execution time is similar to the growth curve for non-dominated sorting time. Compared to fitness evaluation algorithm, non-dominated sorting algorithm contributes to much more portions in the overall NSGA-II computation time in the implementation. Also, besides the two parallelized parts, there are other computations being performed on CPU, such as reproduction process and crowding distance selection. These non-parallelized parts also contribute much more to the increased execution time as population size becomes larger. Despite the large portions of non-parallelized parts, our parallel implementation still shows significant reduction in NSGA-II overall execution time without degrading of final solution quality.

Section 6.2 provides details on this.

Parallel NSGA-II Run Time Complexity

To analyze the speedup of our parallel NSGA-II implementation in terms of algorithm complexity, we analyze the run time complexity for both sequential and parallel portions of NSGA-II summarized in Figure 6.7. Since we only parallelized fitness evaluation and non-dominated sorting parts in this work, the run time complexity analysis only includes times for those two parts. As other portions of NSGA-II algorithm remain sequential in our implementation, they do not contribute to the overall speedup (i.e., they have the similar complexity as the sequential ones, despite some potential differences between the actual implementation of our work and the sequential work in [37]. As the differences are subtle, we are not including these parts in our analysis).

- T(sequential) = T(fitness evaluation) + T(non-dominated sorting)
  = $N*k + M*N^2$
- T(parallel) = T(fitness evaluation) + T(non-dominated sorting) + T(communication)
  = $k + M*N$ + (fitness evaluation: $N*k + k^2$) + (non-dominated sorting: $M*N$)

Figure 6.7: Run Time Complexity for Sequential and Parallel NSGA-II

To clarify, let the size of NSGA-II population $P = N$, size of offspring $Q = N$, graph rank $= k$, number of iterations taken to find all Pareto

fronts = M. For sequential fitness evaluation, each of the N individuals will find the objective cost for each of the k nodes. For sequential non-dominated sorting, we perform the sorting on a combined population of P and Q, so the size will be $2N$. We ignored 2 because it is a constant number. Since we assumed it takes M iterations to find all Pareto front, and in each iteration, each solution in the set of solutions of size N will be compared to every other solutions in the same set. So the pair wise comparisons takes a time of $N*N$. For our parallel fitness evaluation, N (recall N is from $2N$ by ignoring the 2) individuals will perform fitness scores calculation in parallel by N threads, so we only consider the time taken for one thread to do the calculation. Again, the graph size is k, so it takes a thread k times to find all fitness scores for each of the node in the graph. For parallel non-dominated sorting, we use N GPU threads to do the pair wise comparisons simultaneously. In other words, N individuals now perform the comparisons at the same time, which takes a total time of N for all N individuals to finish the comparison. Since we assumed we use M iterations to define all Pareto fronts, so the total time is $M*N$, as in each iteration the number of unselected individuals undergoes comparisons is N at maximal. With parallel NSGA-II, we now have a cost of communication time when

transfer data between CPU and GPU. For fitness evaluation, we have N individuals in population, each has a solution of size k (as it travels all nodes of graph). To compute fitness scores, we need to know the solution of each N individuals. So we use $N*k$ complexity to do this transfer. We also send 2 cost matrices (one for each objective, in our work we focus on bi-objective problems) from CPU to GPU which store the pair wise costs between nodes on the graph. The size of each cost matrix is $k*k$. And we again ignore the constant number 2. For parallel non-dominated sorting, we transfer an unselected population of size maximum of N, and this is done for each of the M iterations to compute all fronts. Therefore the communication cost is $M*N$.

From the two equations in Figure 6.7, we can see the two parallelized parts are faster than the sequential version by an order of magnitude despite of some data transfer overhead incurred when sending/receiving data between CPU and GPU.

6.1.2 Parallel MOEA/D Speedup

For our parallel MOEA/D, Figures 6.8 and 6.9 show the experimental results obtained by experimenting with the three benchmark dataset, with different population sizes and an iteration number 50.

CROSS_MUTATION	Execution Time for Parallel MOEA/D				
Dataset	Population Size	# of Iterations	Average Time (in milliseconds) - parallel	Average Time (in milliseconds) - sequential	Speedup
euclidAB100	50	50	409	4291	1049.14%
	100	50	978	8850	904.91%
euclidAB300	50	50	2564	62498	2437.52%
	100	50	6118	124320	2032.04%
euclidAB500	50	50	7731	252603	3267.40%
	100	50	18498	535060	2892.53%

Figure 6.8: Comparison of Execution Time for Parallel MOEA/D and Sequential MOEA/D

Figure 6.9: Parallel MOEA/D - Execution Time for Various Population Size

Recall that there are two parts being implemented in parallel for MOEA/D: breeding and update of external population. Each part is executed in parallel for all subproblems by a number of CPU threads. The run time complexity for these two parallelized parts and their sequential versions are summarized in Figure 6.10. Within parallel

MOEA/D, there is no internal communication time. In other words, since we used shared memory for thread-level parallelization, each island (i.e., a worker thread) can access the data it needs directly without transfer of information.

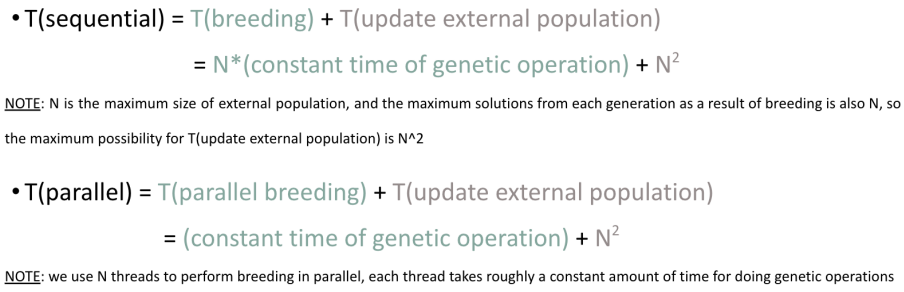

Figure 6.10: Run Time Complexity for Sequential and Parallel MOEA/D

Again, N denotes the size of the population P. For breeding in sequential MOEA/D, the time to perform all genetic operations is constant, and is applied to each of the N individuals in the population. So the total time for breeding is N. To update EP in sequential MOEA/D, since there will be a maximum number of N good solutions returned in each iteration, and the size of EP is always less than or equal to N, so we perform a maximum of $N*N$ comparisons to update the EP. In our parallel breeding, the time for applying genetic operations to each individual is still constant. But now we have N CPU threads to do breeding for all N individuals simultaneously. So the total time for breeding is constant. In parallel MOEA/D, we use same logics to update EP, so

the time complexity is the same as sequential one.

6.1.3 Combined BCE Speedup

For the combined BCE framework with both evolution parties being parallelized, Figure 6.11 and 6.12 show the resulting execution time obtained by running both our framework and the sequential version of BCE. For each of the benchmark dataset, our parallel framework again achieves good result in terms of speedup. When looking at the trend in the increased execution time as a result of increased population size, our parallel BCE shows a relatively steady growth rate for running time when increasing the size of population (figure 6.12).

CROSS_MUTATION	Execution Time for Parallel BCE				
Dataset	Population Size	# of Iterations	Average Time (in milliseconds) - parallel	Average Time (in milliseconds) - sequential	Speedup
	50	50	1410	22098	1567.23%
euclidAB100	100	50	2913	59036	2026.64%
	50	50	6472	169124	2613.16%
euclidAB300	100	50	14226	386572	2717.36%
	50	50	15855	628640	3964.93%
euclidAB500	100	50	35262	1312272	3721.49%

Figure 6.11: Comparison of Execution Time for Parallel BCE and Sequential BCE

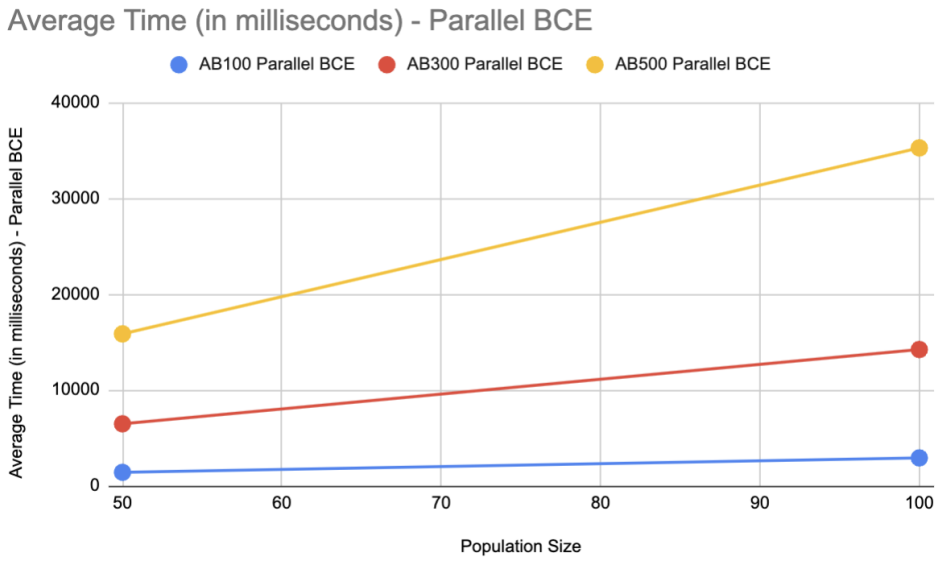

Figure 6.12: Parallel BCE - Execution Time for Various Population Size

The run time complexity for our parallel BCE framework and the sequential one is provided in Figure 6.13. For communication time between NSGA-II and MOEA/D for both sequential and parallel versions (i.e., this is when each evolution part uses good solutions from other evolution partner), it is being neglected in the overall run time as it is minimal.

- T(sequential) = T(NSGA-II) + T(MOEA/D) + T(communication)
  = $N*k + M*N^2$ + N*(constant time of genetic operation) + N^2
- T(parallel) = T(Parallel NSGA-II) + T(Parallel MOEA/D) + T(communication)
  = T(fitness evaluation) + T(non-dominated sorting) + T(communication) + T(parallel breeding) + T(update external population) + T(communication)
  = k + M*N + (fitness evaluation: $N*k + k^2$) + (non-dominated sorting: M*N) + (constant time of genetic operation) + N^2

Figure 6.13: Run Time Complexity for Sequential and Parallel BCE

6.2 Solution Quality

6.2.1 Methods for Checking Solution Quality

In this work, we check the quality of solutions produced by our proposed framework in two ways:

1. Step-by-step Inspection - To ensure the results produced by each step in our framework is reasonable, we first check the results step-by-step manually with debugging mode of our IDE. In other words, we look at the results returned in each stage of the algorithm for each of the parallel NSGA-II, parallel MOEA/D and hybrid BCE implementations.

2. IGD Score Comparisons - Inverted Generational Distance (IGD) scores is a performance indicator that is commonly used when evaluating the performance of multi-objective optimization algorithms in terms of solution quality. Recall when dealing with multi-objective optimizations, we get a set of solutions instead of a single best one. One way to analyze the solution quality in multi-objective scenarios is to calculate IGD scores, in which we do an estimation on how far the distance is between the points in Pareto front produced by our algorithm and the points in true

Pareto front of the given problem. Smaller IGD scores indicate better Pareto front produced by solutions of the algorithm. More formally, IGD is defined by equation 6.1 [37] , where S is the approximated Pareto front produced by the MOP algorithm, P is the optimal Pareto front, $d(x,y)$ is the Euclidean distance between the points x and y, the average minimum distance between S and P is calculated [37].

$$IGD(S,P) = \frac{\sum_{x \in P} min_{y \in S}(d(x,y))}{|P|} \tag{6.1}$$

In our work, we use the known best solutions and its Pareto front obtained from the same open-source bi-objective TSP benchmark library (i.e., [6]) to compute IGD scores for Pareto front produced by our own solutions. We further compare our IGD scores to the scores obtained from sequential BCE framework from [37], and this experiment is conducted for different program configurations.

6.2.2 Experiments and Results

First, we compare the IGD scores produced by the following 6 algorithms: sequential NSGA-II, MOEA/D and BCE; parallel NSGA-II, MOEA/D and BCE. Population size is set to 50 and number of itera-

tions is set to 200. Figure 6.14 shows the results obtained by running experiments on euclidAB100 dataset (i.e., input dataset size of 100 cities).

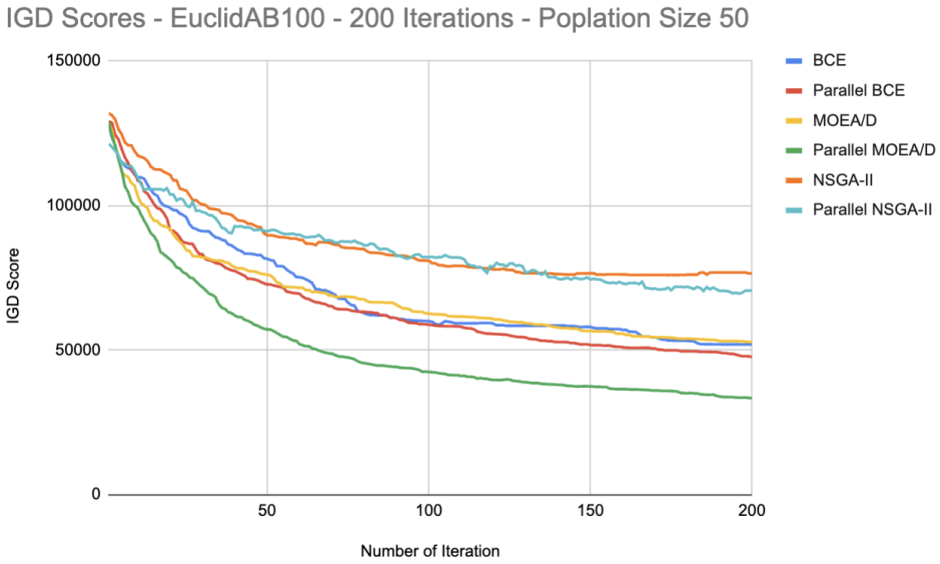

Figure 6.14: IGD Scores for EuclidAB100 with Population Size = 50 and 200 Iterations

From the graph in Figure 6.14, we can see that among the 6 tested algorithms, the parallel MOEA/D and parallel BCE produce the best IGD scores over the same number of iterations under same program settings. The parallel NSGA-II is less optimal because of the randomness involved in the algorithm when it optimizes its solutions set to approach the Pareto front.

6.3 Solution Stability

Once we checked the solution correctness and achieved speedup of our implementation, we looked at the time taken for our proposed implementation to reach solution stability in terms of IGD scores. The benchmark dataset used is euclidAB100 bi-objective TSP dataset. First, we set the number of iterations to 500, with the population size 50. Result is shown in Figure 6.15 .

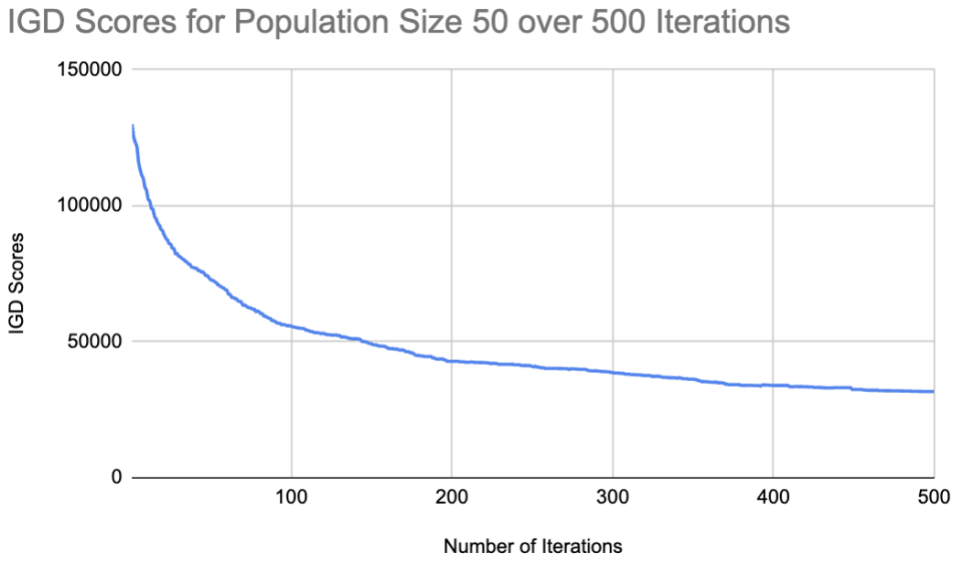

Figure 6.15: IGD Scores for EuclidAB100 with Population Size = 50 over 500 Iterations

From the above graph (Figure 6.15), we can conclude the IGD scores gradually become more stable after 200-250 iterations, with a value smaller than 50,000. Although it is still getting better scores after

200-250 iterations, the curve becomes more stabilized. In contrast, the IGD scores produced in the first 200 iterations are change drastically. The stabilization in better IGD scores occurring in the later iterations we attribute to the search directions becoming more mature due to contributions from both PC and NPC evolutions.

Further, we look at the impact of increasing the population size with a constant number of iterations on resulting IGD scores for a fixed size of input dataset. With euclidAB100 as the input data, we run 200 iterations and record the IGD scores and execution time for each number of iteration. The population sizes used are 50, 100, 200 and 300. Figures 6.16 and 6.17 give details on IGD scores produced by parallel BCE in each iteration with different population sizes. Run time for each number of iterations is also shown. The IGD score between 50,000 and 51,000 and the execution time for each population with different size to reach it are marked in red.

TSPEuclidAB100 TSP_CROSS_MUTATION					120329	117408	115862	111629
Iteration	IGD Scores (Population 50)	Time (milliseconds)	IGD Scores (Population 100)	Time (milliseconds)	IGD Scores (Population 200)	Time (milliseconds)	IGD Scores (POP 300)	Time (milliseconds)
1	131645	272	125849	526	125029	1522	120329	3193
2	125729	307	124519	589	119490	1691	117408	3475
3	122601	331	115808	643	116938	1792	115862	3622
4	119835	355	115390	688	113435	1884	111629	3771
5	115870	380	114829	731	111351	1977	110667	3920
6	113032	407	112729	776	108056	2072	107944	4067
7	109510	432	110955	823	104500	2167	104499	4211
8	105572	457	106571	867	102293	2259	103915	4355
9	104945	481	104148	918	100324	2348	101545	4496
10	103854	505	102281	969	99629.3	2436	97261.9	4639
11	102459	529	99912.7	1012	98699.5	2530	97018.8	4779
12	100404	552	97958.5	1058	96085.9	2623	95360.9	4920
13	97094	577	96848.2	1102	95285.7	2717	94233.9	5064
14	96015	603	95980.1	1145	94643.4	2809	92878.2	5205
15	94392.5	627	94523	1190	92667.1	2901	92287.3	5350
16	93288.6	650	93600.8	1234	91230.5	3030	91502.8	5493
17	91975	674	90241	1285	90812.1	3122	89485.8	5640
18	91063.3	699	90159.8	1336	88832.5	3217	88126.7	5789
19	90530	723	88760	1380	88001.3	3312	87229	5933
20	89270.2	747	88229	1423	87456	3402	85249.8	6075
21	88634.3	771	87418	1467	83899.3	3493	84934.9	6219
22	87592.7	794	86394.7	1512	83571.7	3582	83861.4	6362
23	87106.2	817	84273.2	1557	82022.6	3670	82959.1	6504
24	86455.8	841	83888.5	1602	81918.1	3758	81937.5	6650
25	85795.2	865	82048.2	1652	81138.6	3846	81642.1	6790
26	83936.9	890	81659.3	1695	80304.5	3940	80031.2	6936
27	83569.3	913	79768.5	1739	79398.8	4030	79957.5	7079
28	82991.8	937	79411.1	1783	78430.2	4121	78400.6	7220
29	82522.1	961	78735.2	1826	77428.1	4245	78335.2	7370
30	81901.5	984	77416.1	1870	76591.5	4334	77468.8	7518
31	80930	1007	76568	1914	76072.7	4426	77201.1	7669
32	80452.9	1030	75584.7	1957	74639.7	4519	76200.7	7815
33	79885.1	1053	75095.7	2000	74233.2	4610	75394.7	7963
34	79570.5	1075	73907.7	2044	73806.6	4700	74854	8114
35	79326.8	1097	73182	2088	71828.1	4823	74306.2	8269
36	78950.2	1121	71957.8	2130	71761.9	4912	73661.7	8427
37	78387	1143	71761.8	2173	70719.8	5002	73059.4	8584
38	78270.2	1166	71727.8	2215	70677.6	5092	72377.2	8743
39	77926.7	1189	70904.2	2258	69790.9	5181	71805.9	8898
40	77781.5	1213	70708.2	2301	69723.7	5274	70667.5	9057
41	76720.1	1235	69771.2	2344	69044.3	5362	70282	9218
42	76556.6	1257	69199.6	2385	68885.8	5453	69934.5	9376
43	75642.6	1278	69063.1	2428	68342.9	5543	69292.8	9531
44	75416	1301	67423	2471	68076.4	5633	68661.4	9691
45	73733.9	1326	67276.5	2522	67189.1	5723	68358	9855
46	73722.4	1349	66893.7	2573	66175.7	5816	67555.3	10018
47	72786.6	1373	66225.8	2617	65954.2	5905	66876.9	10181
48	72164.6	1396	65939.6	2668	65312.8	5995	66697.4	10350
49	71737.3	1419	65535.6	2712	64909	6084	66195.8	10518
50	71406.9	1443	65296.9	2755	64521.3	6172	65625.6	10683
51	71283.2	1466	64981.3	2805	64388.6	6259	64704.5	10846
52	70978.2	1489	64204.8	2850	63820.8	6346	63943.3	11005
53	70696	1512	64003.5	2894	63544.9	6436	63286.6	11169
54	70333.9	1536	63220.2	2939	63050.7	6528	62668.9	11331
55	70170.4	1559	63214.8	2982	62178.5	6654	62453.4	11489
56	69523.4	1582	62892.2	3025	61550.3	6748	62059.2	11648
57	69352.4	1606	62510.4	3069	60979.7	6840	61565.1	11806
58	69241.2	1629	62289.7	3112	60496.1	6933	60809.9	11968
59	68577.6	1652	61834.1	3158	60345.5	7064	60420.8	12127
60	68541.7	1678	61516	3203	60246.9	7156	60281.9	12295
61	68063.6	1702	60861.9	3248	59854	7250	59790.8	12454
62	67739.9	1728	60686.2	3292	59599.5	7345	59521	12619
63	67723.2	1753	60239.6	3337	59193.5	7438	58909	12782
64	67520.2	1777	60091.2	3381	59028.3	7535	58345.1	12947
65	66570.7	1802	59861.7	3425	58878.1	7629	58019.5	13108
66	66246.1	1828	59584.1	3471	58233.9	7724	57258	13278
67	65723.7	1853	59466.5	3517	58165.5	7854	56960.9	13449
68	65103.1	1877	59174	3562	57809.3	7957	56433.2	13615
69	64341.9	1900	58903.4	3608	57315.6	8056	55998.1	13773
70	64140.9	1924	58751.6	3653	56899	8155	55692.4	13935
71	64008.6	1951	58321.6	3697	56721.7	8253	55348.7	14096
72	63777.2	1976	58264.8	3743	56617.2	8351	54889.9	14260
73	63737	2002	57178	3787	56367.3	8453	53947.4	14418
74	63164.1	2028	57154.7	3830	56088.6	8586	53906.9	14578
75	62644.1	2052	56945	3873	55904.6	8684	53193.9	14730
76	62573.1	2077	56314.2	3918	54941.1	8784	53125.6	14886
77	62434.8	2103	56234	3962	54442	8884	52352.7	15042
78	62322.7	2129	55502.4	4013	53876.6	8986	52175.4	15200
79	62021.2	2153	55493	4065	53566.8	9084	51905.8	15355
80	61972.8	2176	55485.3	4112	53423.6	9178	51779.3	15511
81	61637	2199	54893.3	4156	52724	9274	51344.9	15666
82	61184.5	2223	54876.5	4200	52415	9375	51333.8	15826
83	61085.5	2246	54735.1	4243	52352	9471	50973.3	15981
84	61015	2269	54664.4	4287	51960.5	9570	50763.2	16140
85	60939.7	2292	54195.1	4331	51869.4	9673	50616	16298
86	60939.7	2315	53985.7	4375	51744.5	9774	50215.6	16457
87	60722.9	2339	53848.8	4420	51602.4	9874	50077.4	16613
88	60546.4	2366	53191.1	4464	51371.5	9977	49923.9	16777
89	60493.1	2391	52750.4	4515	51296	10074	49580.1	16936
90	60347.9	2417	52588.6	4559	51060.8	10177	49540.5	17095
91	59699.1	2440	52582.3	4609	50874.6	10276	48957.4	17254
92	59397.3	2466	52471.9	4653	50836	10376	48646.2	17409
93	58398.3	2492	51960.5	4698	50159.4	10479	48537.9	17565
94	58378.5	2516	51497.3	4743	49961.3	10579	48124.1	17717
95	57988.1	2539	51484.6	4787	49621	10680	47946.4	17871
96	57983.7	2565	51062.8	4831	49342.2	10780	47533.8	18028
97	57676.4	2589	50704.1	4876	49132.8	10882	47181.3	18186
98	56940.1	2612	50465.8	4921	49103.2	10979	46978.1	18344
99	56936.4	2635	49854	4964	48980.8	11079	46898.6	18501
100	56454.2	2659	49757.5	5009	48883.5	11176	46595	18654

Figure 6.16: IGD Scores with Increasing Population Size Part 1

101	56394	2683	49561.7	5054	48805.8	11308	46504.5	18811
102	56361.5	2707	49543.8	5098	48568.1	11411	46378.4	18969
103	56021.6	2730	49422.3	5145	48499.5	11515	46270.4	19123
104	55877.3	2754	49356.6	5191	48273.2	11614	45651.4	19277
105	55892.8	2778	49286.9	5236	48197.5	11712	45277	19432
106	54959	2801	49093.9	5283	48147.2	11809	45041	19584
107	54932.1	2824	49093.9	5329	47816.9	11912	44882.5	19732
108	54932.1	2846	48924.3	5376	47476.7	12010	44756.7	19883
109	54928.2	2868	48687.2	5423	47418.1	12128	44147.2	20033
110	54439.6	2891	48156.6	5468	47373.5	12269	44003.2	20192
111	54438.9	2913	47560.5	5515	46991.4	12394	43887.3	20345
112	53151.7	2937	47422	5561	46707.4	12513	42747.1	20495
113	52968.9	2959	47236.6	5606	46574.9	12625	42687.3	20644
114	52345.8	2982	46829.3	5652	46472.1	12738	42579	20793
115	51881.1	3004	46654	5697	45917.9	12834	42365.6	20950
116	51868.9	3027	46424.1	5742	45901.3	12931	42351.3	21106
117	51743.9	3048	46335.7	5786	45717.5	13028	42152.4	21262
118	51158.2	3070	45510.2	5830	45533.5	13126	41839.2	21417
119	50942.1	3092	45385.8	5874	45273.4	13224	41645	21572
120	50515.6	3114	45213.5	5917	45167	13324	41539.6	21723
121	50038.3	3137	44943.6	5963	44973.6	13430	41140.7	21875
122	50038.3	3160	44886.6	6009	44785.1	13566	41107.4	22032
123	50038.3	3182	44814	6054	44682.2	13680	41000.8	22184
124	50015.6	3206	44752	6098	44383.6	13826	40866.1	22336
125	49759.8	3229	44666.5	6143	44269.3	13986	40490.8	22488
126	49759.8	3252	44621.5	6189	44034.6	14105	40259.2	22638
127	48950.2	3274	44503.2	6234	43773.2	14212	40012.8	22792
128	48950.2	3295	44312.2	6279	43741.8	14321	39935.3	22947
129	48893.4	3318	44312.1	6324	43691.6	14431	39898.7	23101
130	48767.9	3340	44243.7	6370	43518.7	14531	39678.6	23256
131	48767.7	3362	44058.7	6414	43399	14628	39594.6	23412
132	47752.4	3384	44058.7	6460	43291.2	14764	39570	23567
133	47648	3405	43961.9	6505	43112.5	14862	39294.6	23725
134	47576.1	3427	43946.1	6550	43032.5	14962	39044.9	23884
135	47546.9	3449	43820.8	6596	42614.4	15064	38902.8	24041
136	46768.8	3471	43794.9	6642	42646.2	15162	38837.1	24201
137	46712.2	3494	43235.2	6688	42560.5	15260	38815.6	24360
138	46711.8	3517	43204.1	6734	42329.9	15357	38677.6	24522
139	46656.6	3541	43049.8	6780	42206	15462	38444.3	24697
140	46402.8	3564	43037.9	6825	42158.9	15562	38364.3	24863
141	46257.7	3586	42972.1	6871	41905	15659	38298.7	25021
142	46145.8	3609	42932.1	6917	41874.7	15757	38242.8	25183
143	46145.8	3633	42895.4	6964	41759.3	15860	38039.1	25337
144	46135.3	3656	42821.4	7017	41735.7	15991	37781.9	25488
145	45958	3679	42561.4	7066	41425.7	16095	37572.8	25646
146	45960.4	3702	42566.4	7113	41279.6	16197	37508.8	25800
147	45893.9	3726	42530.3	7161	41101.9	16298	37067	25951
148	45332.1	3747	42421.4	7208	40998.5	16398	36998.5	26120
149	45308	3769	42059.2	7255	40881.4	16498	36884.5	26275
150	45198.3	3792	41988.7	7303	40847.3	16632	36835.6	26432
151	45213.1	3814	41874.8	7351	40539.2	16733	36758.9	26589
152	44938.3	3837	41835.8	7401	40533.1	16866	36703.8	26742
153	44703.8	3859	41635.9	7452	40492.5	16965	36570.3	26894
154	44703.8	3882	41313.5	7500	40349.4	17065	36482.1	27054
155	44467.3	3905	40982.8	7549	40262.1	17168	36424	27207
156	44467.3	3928	40966.9	7600	40223.4	17265	36410.4	27358
157	43903	3951	40842.3	7648	39851.6	17368	36375.5	27508
158	43884.4	3974	40834.3	7696	39686.1	17467	36295.4	27661
159	43736.9	3997	40793.5	7743	39545.6	17564	36148.6	27815
160	43735.8	4020	40700.6	7791	39447.1	17667	36141	27970
161	43465.1	4043	40699.9	7838	39244.6	17766	35940.3	28128
162	43496.8	4066	40472.3	7885	39079.1	17867	35895.4	28286
163	43496.8	4089	40375	7933	39074.8	17973	35840.6	28441
164	43494.6	4111	40355.7	7981	39074.8	18073	35699.7	28603
165	43494.6	4135	40315	8031	38791.6	18174	35537.6	28760
166	43458.9	4160	40235.3	8080	38709.6	18273	35422.6	28921
167	43433.4	4183	40075.7	8129	38676.5	18372	35316.9	29076
168	43433.4	4206	39990.1	8177	38565.2	18472	35180	29233
169	43390.7	4231	39855.1	8226	38489.7	18572	35113.9	29393
170	43376.6	4255	39358.3	8271	38433.9	18670	34784.2	29548
171	42887.6	4280	39281.1	8316	38235.5	18769	34718	29700
172	42565.3	4305	39277.7	8363	38220.4	18864	34659.8	29852
173	42565.3	4332	39165.9	8409	38176.6	18960	34643.9	30012
174	42850.6	4355	39081	8454	38177.9	19058	34614	30167
175	42836.3	4379	39000.7	8500	38004.5	19154	34562.1	30326
176	42672.3	4402	38888.9	8554	37661.7	19250	34525.6	30483
177	42672.3	4428	38847.7	8607	37523.4	19383	34476.8	30638
178	42194.8	4453	38702.8	8653	37479.6	19480	34299.4	30790
179	42189.3	4478	38562.2	8699	37416.3	19614	34237.2	30943
180	42189.3	4501	38556.3	8751	37304.2	19714	34143.1	31097
181	42172.5	4525	38411.2	8799	37168.1	19815	34030.1	31256
182	42169.2	4550	38399.5	8846	37168.1	19911	33951.4	31413
183	42169.2	4575	38182.3	8892	37088.8	20010	33886.5	31563
184	41996.3	4601	38095.5	8944	37041.4	20108	33826.9	31722
185	41940.3	4625	38059.1	8996	36737.1	20208	33499.2	31877
186	41900.5	4648	38059.1	9049	36658.2	20304	33464.3	32030
187	41898.8	4672	37920.5	9095	36839	20437	33352.8	32184
188	41691.5	4694	37819.2	9141	36687.6	20534	33324.3	32342
189	41691.5	4717	37788.6	9186	36601.9	20635	33218.5	32495
190	41691.5	4740	37669.7	9233	36200	20737	33115.6	32649
191	41836.9	4764	37581.7	9281	35984.7	20835	33003.9	32809
192	41693.8	4787	37536.2	9327	35984.6	20937	32974.3	32968
193	41686.3	4809	37548.5	9373	35908	21037	32940.7	33126
194	41371.7	4832	37520.5	9419	35821.6	21136	32941.6	33284
195	41259.2	4855	37397.8	9469	35779.9	21236	32869	33441
196	41130.2	4877	37378.3	9518	35748.4	21335	32664	33597
197	41130.2	4900	37380.9	9568	35784	21432	32559.2	33759
198	40883.5	4923	37340.7	9617	35736.7	21531	32557.3	33918
199	40904.8	4946	37340.7	9666	35681.1	21630	32373.2	34073
200	40738.7	4969	37328.3	9715	35681.1	21730	32240.1	34228

Figure 6.17: IGD Scores with Increasing Population Size Part 2

From the table (Figure 6.16 and 6.17), we can see it takes less number of iterations for population with larger sizes to obtain the same IGD scores compared to small-sized population, but the execution time

is increased. Figure 6.18 and 6.19 shows this property.

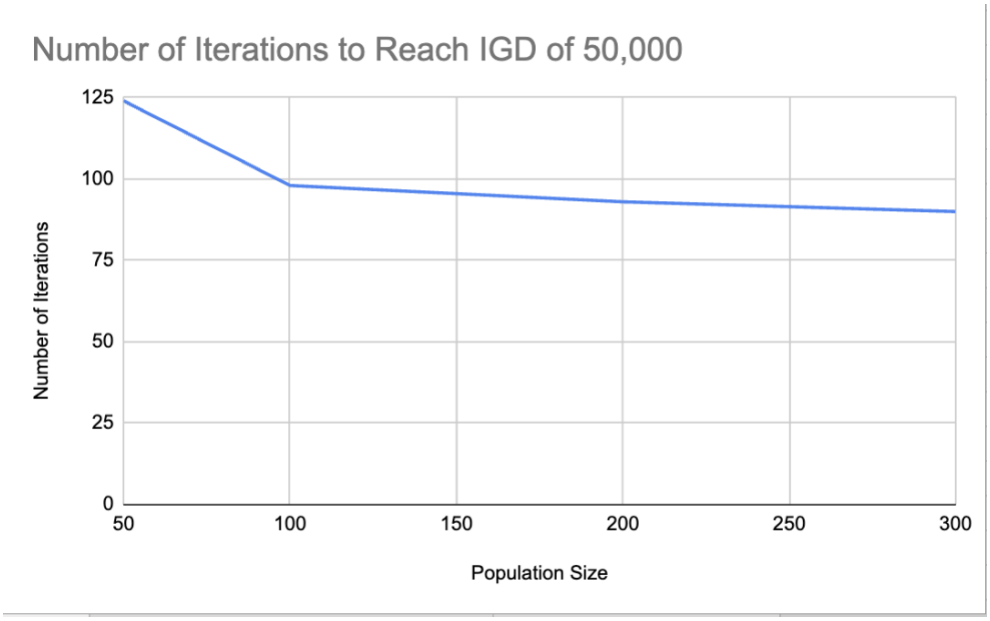

Figure 6.18: Number of Iterations for Different Sized Population to Reach IGD of 50,000

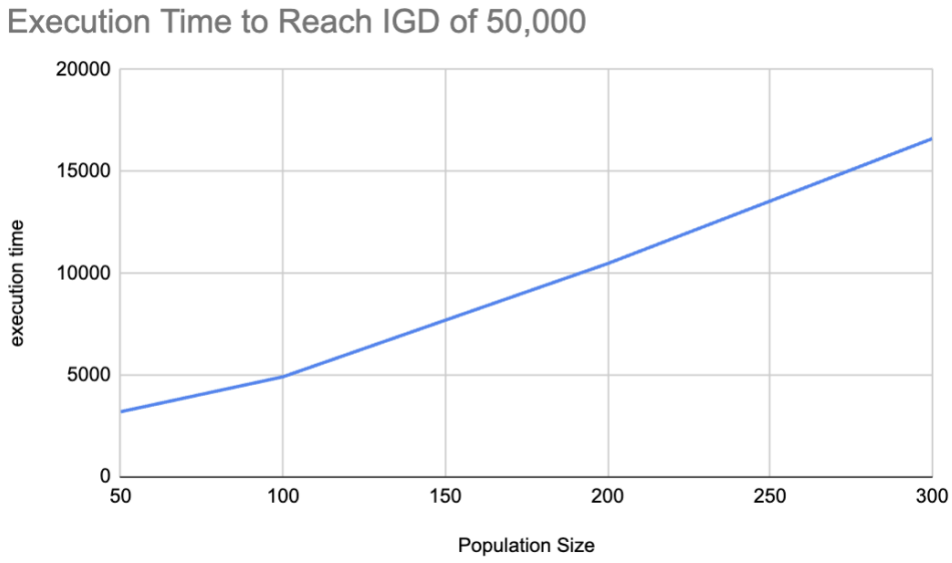

Figure 6.19: Execution Time for Different Sized Population to Reach IGD of 50,000

Despite the increased execution time as population size grows, our

proposed framework is still faster in getting to the same IGD scores compared to the sequential BCE as shown in section 6.1, as parts of each evolution are now being executed in parallel.

6.4 Efficiency and Scalability

Recall for each parallelized part (parallel fitness evaluation and non-dominate sorting algorithms) of NSGA-II, we utilize GPU threads to distribute the workload. More precisely, for a population of size N, the number of GPU threads we used in our computation is $2N$ (since for NSGA-II algorithm, for each population of size N, an offspring population also of size N is created and involved in evolution process. And we use 1 GPU thread to handle calculations for 1 individual in overall population). Therefore, the number of GPU threads we used each time is as twice the number as the population size. For example, when population size is 500, we used 1000 GPU threads. Note, that on our GPU device used in this work, the maximum number of threads per block is 1024. Therefore in case of population size of 500, 1 GPU block is used. For population size of 1000, we used 2000 GPU threads, which is 2 GPU blocks on our device. Since the maximum number of threads in each block is 1024 in our machine, and 2 blocks has 2048 threads. For

a population size of 1000, 2000 threads is used across 2 blocks. But 48 threads in the second block are wasted. And when population is 500, we use 1000 threads, with 24 threads being wasted. When comparing the execution time between 500 population and 1000 population, the time increase is as expected, but with a lower growth rate as the workload are spread over 2 blocks than 1. Figure 6.20 shows this. The busier we keep the GPU the more stable is the growth rate. The larger the population size, the more blocks are needed and scheduled on the cores. However, there is a limitation on how much we can increase the population size on a single GPU machine.

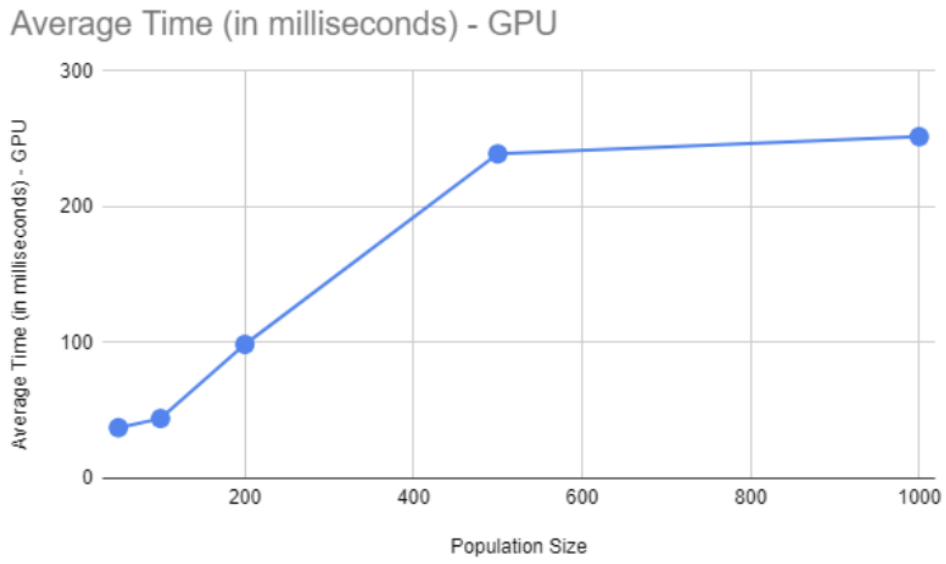

Figure 6.20: Steady Growth Curve for Execution Time with 2 GPU blocks when Population Size Larger than 500

We hypothesize that if we have had more than one GPU, we could

experiment with larger population sizes and increase the efficiency of the parallel implementation and thereby increase scalability.

For parallel MOEA/D, since we are using CPU threads to achieve parallelism, the scalability of our program is bounded by available threads in the computing device. For experiments performed on our CPU device, only 12 threads can run simultaneously. This limits the size of the population. To increase scalability, we need more CPU cores and shared memory to handle larger problem sizes.

Chapter 7

Conclusion and Future Works

7.1 Conclusion

In this thesis, we developed, implemented and analyzed a parallelized hybrid algorithm for solving MOPs based on the sequential BCE framework [37]. The framework is composed of two parallel evolutionary processes: PC evolution and NPC evolution. Each component follows its own evolutionary process with individuals in each population evolving simultaneously. NSGA-II is chosen as the PC evolutionary algorithm while MOEA/D, the NPC evolutionary algorithm. Each algorithm has its own strength and weaknesses. The hybrid algorithm using a collaborative and co-evolutionary approach, uses the strengths of these algorithms to produce a fast, convergent and diverse solutions. As the hybrid algorithm is compute intensive, we parallelized the hy-

brid algorithm taking into consideration the suitability of the architecture for the algorithm. While some of the modules in NSGA-II are data parallel and suitable for accelerating on the GPU, the modules in MOEA/D are compute and communication intensive, making it suitable for shared memory CPU-based machines. However, to allow for more scalability, we considered an island model, where each island executes the MOEA/D algorithm. The islands exchange information to provide more diverse solutions. The solutions in MOEA/D and NSGA-II are exchanged periodically to move towards a better solution. Experiments were conducted on the algorithms individually and hybrid on benchmark dataset. Using the IGD scores, we evaluated the parallel algorithm against the sequential algorithm. The scores indicate the hybrid parallel algorithm and parallel MOEA/D performs better than the other four tested algorithms (NSGA-II, MOEA/D, BCE; parallel NSGA-II) giving it a more efficient algorithm. The GPU architecture is a fine grained architecture and therefore, the more fine-grained computations are executed on the machine, the better the performance. However, if the threads encountered conditional statements, such as in the non-dominating sorting algorithm, the performance degrades. Overall the parallel hybrid algorithm produces better performance compared to a

sequential algorithm.

7.2 Future Work

In this work, we only considered NSGA-II for PC evolution and MOEA/D for NPC evolution. There are other EAs for MOPs that can also be considered in the hybrid framework. Further, we only used IGD metrics when looking at the quality of our solutions. It confirms the good performance with an emphasis on convergence. Other metrics can be considered for evaluating the diversity of the results. The decomposition strategy for MOEA/D used in this thesis is the weighted sum approach. In the future, any other decomposition strategy can be tested. The parallelization is on one GPU machine. This can be extended to multiple GPU machines for more scalability. Furthermore, the island model topology for MOEA/D can be investigated. Some topologies may behave better than others (centralized star versus distributed ring). Lastly, the BCE framework can be applied to dynamic optimization problems. It is not clear how the parallel algorithm would affect dynamic scenarios. This would be interesting to study in the future.

Bibliography

[1] Distributed memory systems.

[2] *An Introduction to Genetic Algorithms*. MIT Press, Cabridge, MA.

[3] Multithreading cpp.

[4] Nvidia developer.

[5] Scalability lecture.

[6] Tsp data.

[7] Shared memory systems. *High performance computing*, 2018.

[8] Distributed computing. *Modeling of resistivity and acoustic borehole logging measurements using finite element methods*, 2021.

[9] Cuda c programming guide, Oct 2023.

[10] Sandro Artina, Cristiana Bragalli, Giovanni Erbacci, Angela Marchi, and Marzia Rivi. Nsga-ii in optimal design of water dis-

tribution networks. *Journal of Hydroinformatics*, 14:310–323, 04 2012.

[11] B.W. Bader. Constrained and unconstrained optimization. *Comprehensive Chemometrics*, page 507–545, 2009.

[12] M. Basseur and E. K. Burke. Indicator-based multi-objective local search. In *2007 IEEE Congress on Evolutionary Computation*, pages 3100–3107, 2007.

[13] A. Bhardwaj and A. Tiwari. Breast cancer diagnosis using genetically optimized neural network model. *Expert Systems with Applications*, 42:4611–4620, 2015.

[14] Erick Cantú-Paz and David E. Goldberg. Efficient parallel genetic algorithms: theory and practice. *Computer Methods in Applied Mechanics and Engineering*, 186(2):221–238, 2000.

[15] Lucas Braga de Oliveira, Carolina G. Marcelino, Anolan Milanés, Paulo E. M. Almeida, and Leonel M. Carvalho. A successful parallel implementation of nsga-ii on gpu for the energy dispatch problem on hydroelectric power plants. In *2016 IEEE Congress on Evolutionary Computation (CEC)*, pages 4305–4312, 2016.

[16] Murilo Zangari de Souza and Aurora Trinidad Pozo. Parallel

moea/d-aco on gpu. *Advances in Artificial Intelligence – IBERAMIA 2014*, page 405–417, 2014.

[17] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[18] Kalyan Deb. *Multiobjective Optimization Using Evolutionary Algorithms. Wiley, New York*. 01 2001.

[19] Juan J. Durillo, Antonio J. Nebro, Francisco Luna, and Enrique Alba. A study of master-slave approaches to parallelize nsga-ii. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, 2008.

[20] Juan J. Durillo, Qingfu Zhang, Antonio J. Nebro, and Enrique Alba. Distribution of computational effort in parallelnbsp;moea/d. *Lecture Notes in Computer Science*, page 488–502, 2011.

[21] Jesús Guillermo Falcón-Cardona, Raquel Hernández Gómez, Carlos A. Coello Coello, and Ma. Guadalupe Castillo Tapia. Parallel multi-objective evolutionary algorithms: A comprehensive survey. *Swarm and Evolutionary Computation*, 67:100960, 2021.

[22] Xiaodan Gao, Bingzhen Chen, Xiaorong He, Tong Qiu, Jichun Li,

Chongming Wang, and Longjiang Zhang. Multi-objective optimization for the periodic operation of the naphtha pyrolysis process using a new parallel hybrid algorithm combining nsga-ii with sqp. *Computers amp;amp; Chemical Engineering*, 32(11):2801–2811, 2008.

[23] Samarth Gupta and Gary Tan. A scalable parallel implementation of evolutionary algorithms for multi-objective optimization on gpus. In *2015 IEEE Congress on Evolutionary Computation (CEC)*, pages 1567–1574, 2015.

[24] Tomohiro Harada and Enrique Alab. Parallel genetic algorithms: A useful survey. *ACM Computing Surveys*, 53:1–39, 2020.

[25] Le Huy Hoang, Nguyen Viet Long, Nguyen Ngoc Thu Phuong, Ho Minh Hoang, and Quan Thanh Tho. *Towards Parallel NSGA-II: An Island-Based Approach Using Fitness Redistribution Strategy", bookTitle="Soft Computing: Biomedical and Related Applications*, pages 183–200. Springer International Publishing, Cham, 2021.

[26] Dario Izzo, Marek Ruciński, and Francesco Biscani. The gener-

alized island model. *Parallel Architectures and Bioinspired Algorithms*, page 151–169, 2012.

[27] Siwei Jiang, Jie Zhang, Yew-Soon Ong, Allan N. Zhang, and Puay Siew Tan. A simple and fast hypervolume indicator-based multiobjective evolutionary algorithm. *IEEE Transactions on Cybernetics*, 45(10):2202–2213, 2015.

[28] Yi-Tung Kao and Erwie Zahara. A hybrid genetic algorithm and particle swarm optimization for multimodal functions. *Applied Soft Computing*, 8(2):849–857, 2008.

[29] Liangjun Ke, Qingfu Zhang, and Roberto Battiti. Moea/d-aco: A multiobjective evolutionary algorithm using decomposition and antcolony. *IEEE Transactions on Cybernetics*, 43(6):1845–1859, 2013.

[30] B D Latter. The island model of population differentiation: A general solution. *Genetics*, 73(1):147–157, 1973.

[31] Hui Li and Qingfu Zhang. Multiobjective optimization problems with complicated pareto sets, moea/d and nsga-ii. *IEEE Transactions on Evolutionary Computation*, 13(2):284–302, 2009.

[32] Miqing Li, Shengxiang Yang, and Xiaohui Liu. Pareto or non-

pareto: Bi-criterion evolution in multiobjective optimization. *IEEE Transactions on Evolutionary Computation*, 20(5):645–665, 2016.

[33] Xiaolong Li, Zhecong Zhang, Wei Sun, Yang Liu, and Jiafu Tang. Parallel dynamic nsga-ii with multi-population search for rescheduling of seru production considering schedule changes under different dynamic events. *Expert Systems with Applications*, 238:121993, 2024.

[34] Zheng Li, Yi Bian, Ruilian Zhao, and Jun Cheng. A fine-grained parallel multi-objective test case prioritization on gpu. *Search Based Software Engineering*, page 111–125, 2013.

[35] Weiduo Liao, Hisao Ishibuchi, Lie Meng Pang, and Ke Shang. Parallel implementation of moea/d with parallel weight vectors for feature selection. In *2020 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 1524–1531, 2020.

[36] Ying Ying Liu. *Multi-objective optimization on dynamic complex networks*. PhD thesis, University of Manitoba, 2022.

[37] Ying Ying Liu, Parimala Thulasiraman, and Nelishia Pillay. Bi-criterion coevolution for the multi-objective travelling salesperson

problem. In *2022 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8, 2022.

[38] Andrea Mambrini and Dario Izzo. Pade: A parallel algorithm based on the moea/d framework and the island model. *Parallel Problem Solving from Nature – PPSN XIII*, page 711–720, 2014.

[39] Alejandro Martín, Raúl Lara-Cabrera, Félix Fuentes-Hurtado, Valery Naranjo, and David Camacho. Evodeep: A new evolutionary approach for automatic deep neural networks parametrisation. *Journal of Parallel and Distributed Computing*, 117:180–191, 2018.

[40] Kaisa Miettinen. Nonlinear multiobjective optimization. *International Series in Operations Research amp;amp; Management Science*, 1998.

[41] Antonio J. Nebro and Juan J. Durillo. A study of the parallelization of the multi-objective metaheuristic moea/d, Jan 1970.

[42] Eneko Osaba, Esther Villar-Rodriguez, Javier Del Ser, Antonio J. Nebro, Daniel Molina, Antonio LaTorre, Ponnuthurai N. Suganthan, Carlos A. Coello Coello, and Francisco Herrera. A tutorial on the design, experimentation and application of metaheuristic

algorithms to real-world optimization problems. *Swarm and Evolutionary Computation*, 64:100888, 2021.

[43] Florina Roxana Padurariu and Cristina Marinescu. Nsga-ii: Implementation and performance metrics extraction for cpu and gpu. In *2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 494–499, 2014.

[44] Petr Pospichal, Jiri Jaros, and Josef Schwarz. Parallel genetic algorithm on the cuda architecture. In Cecilia Di Chio, Stefano Cagnoni, Carlos Cotta, Marc Ebner, Anikó Ekárt, Anna I. Esparcia-Alcazar, Chi-Keong Goh, Juan J. Merelo, Ferrante Neri, Mike Preuß, Julian Togelius, and Georgios N. Yannakakis, editors, *Applications of Evolutionary Computation*, pages 442–451, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[45] purdue. Multi-objective - purdue university college of engineering.

[46] T. P. Karnowski S.-H. Lim S. R. Young, D. C. Rose and R. M. Patton. Optimizing deep learning hyper-parameters through an evolutionary algorithm. In *Proceedings of the Workshop on Machine Learning in High-Performance Com- puting Environments*, pages 1–5, 2015.

[47] Yuji Sato, Mikiko Sato, Mads Midtlyng, and Minami Miyakawa. Parallel and distributed moea/d with exclusively evaluated mating and migration. In *2020 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8, 2020.

[48] Karthik Sindhya, Kaisa Miettinen, and Kalyanmoy Deb. A hybrid framework for evolutionary multi-objective optimization. *IEEE Transactions on Evolutionary Computation*, 17(4):495–511, 2013.

[49] Marco Tomassini. *Spatially Structured Evolutionary Algorithms: Artificial Evolution in Space and Time (Natural Computing Series)*. Springer-Verlag, Berlin, Heidelberg, 2005.

[50] Xuewen Xia, Huixian Qiu, Xing Xu, and Yinglong Zhang. Multi-objective workflow scheduling based on genetic algorithm in cloud environment. *Information Sciences*, 606:38–59, 2022.

[51] Lingxi Xie and Alan Yuille. Genetic cnn. In *IEEE International Conference on Computer Vision (ICCV)*, pages 1388–1397, 2017.

[52] Cheng-Jin Ye and Min-Xiang Huang. Multi-objective optimal power flow considering transient stability based on parallel nsga-ii. *IEEE Transactions on Power Systems*, 30(2):857–866, 2015.

[53] Weiqin Ying, Yuehong Xie, Yu Wu, Bingshen Wu, Shiyun Chen,

and Weipeng He. Universal partially evolved parallelization of moea/d for multi-objective optimization on message-passing clusters. *Soft Computing*, 21(18):5399–5412, 2016.

[54] Jusheng Yu, Lu Li, and YuTao Qi. Parallel moea/d for real-time multi-objective optimization problems. *E-Learning and Games*, page 236–240, 2019.

[55] P.Th. Zacharia and N.A. Aspragathos. Optimal robot task scheduling based on genetic algorithms. *Robotics and Computer-Integrated Manufacturing*, 21(1):67–79, 2005.

[56] Qingfu Zhang and Hui Li. Moea/d: A multiobjective evolutionary algorithm based on decomposition. *IEEE Transactions on Evolutionary Computation*, 11(6):712–731, 2007.