Scalable Parallel Evolutionary Optimisation based on High Performance Computing

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

Chen Jin

BEng(ElecEng), Beihang University, China
MEng(ElecEng), Beihang University, China

School of Science

College of Science, Engineering, and Health

RMIT University

June 2019

Declaration

I certify that except where due acknowledgement has been made, the work is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program; any editorial work, paid or unpaid, carried out by a third party is acknowledged; and, ethics procedures and guidelines have been followed.

Chen Jin
School of Science
College of Science, Engineering, and Health
RMIT University
29 June 2019

Acknowledgement

I would like to express my special appreciation and thanks to my supervisors Dr. Jeffrey Chan and Dr. Kai Qin, not only for their tremendous support to my thesis writing, but also for the enlightenment on academic thinking which is the lifelong benefit. Without their consistent and illuminating instruction, this thesis could not have reached its present form.

I want to say thanks to my wife Bichen Ni, who has been doing the utmost to support me in all these years since she has come to Australia. I owe her everything. I am also hugely appreciative to my beloved dad and mum for their loving considerations and great confidence in me all through these years.

I also want to say thanks to Dr. Xiaodong Li and Dr. Andy Song, who gave me plenty of precious advice on my research. I also learn a lot from the weekly ECML meeting organised by them. Moreover, I owe my sincere gratitude to my friends Pengfei Li, Boyu Zhang, Youhan Xia, Hui Song, Pei-Wei Tsai, Tsz Ho Wong, Wei Shao, Steven Wu and Xiaolu Lu. They gave me their help and time in listening to me and helping me work out my problems during the difficult course of the thesis.

I acknowledge the support of powerful computing facilities by National Computational Infrastructure (NCI). I also acknowledge the RMIT School of Graduate Research and RMIT School of Science for their support in the form of scholarship, research support and grants.

Credits

Portions of the materials used in this thesis have previously appeared or under consideration in the following scientific publications:

- Chen. Jin, A. K. Qin and Ke Tang, "Local ensemble surrogate assisted crowding differential evolution," in Evolutionary Computation (CEC), 2015 IEEE Congress on. IEEE, 2015, pp. 433-440.
- Chen Jin and A. K. Qin, "A GPU-based Implementation of Brain Storm Optimization," in Evolutionary Computation (CEC), 2017 IEEE Congress on. IEEE, 2017, pp. 2698-2705.

Abstract

Evolutionary algorithms (EAs) have been successfully applied to solve various challenging optimisation problems. Due to their stochastic nature, EAs typically require considerable time to find desirable solutions; especially for increasingly complex and large-scale problems. As a result, many works studied implementing EAs on parallel computing facilities to accelerate the time-consuming processes. Recently, the rapid development of modern parallel computing facilities such as the high performance computing (HPC) bring not only unprecedented computational capabilities but also challenges on designing parallel algorithms. This thesis mainly focuses on designing scalable parallel evolutionary optimisation (SPEO) frameworks which run efficiently on the HPC.

Motivated by the interesting phenomenon that many EAs begin to employ increasingly large population sizes, this thesis firstly studies the effect of a large population size through comprehensive experiments. Numerical results indicate that a large population benefits to the solving of complex problems but requires a large number of maximal fitness evaluations (FEs). However, since sequential EAs usually requires a considerable computing time to achieve extensive FEs, we propose a scalable parallel evolutionary optimisation framework that can efficiently deploy parallel EAs over many CPU cores at CPU-only HPC. On the other hand, since EAs using a large number of FEs can produce massive useful information in the course of evolution, we design a surrogate-based approach to learn from this historical information and to better solve complex problems. Then this approach is implemented in parallel based on the proposed scalable parallel framework to achieve remarkable speedups.

Since demanding a great computing power on CPU-only HPC is usually very expensive, we design a framework based on GPU-enabled HPC to improve the cost-effectiveness of parallel

EAs. The proposed framework can efficiently accelerate parallel EAs using many GPUs and can achieve superior cost-effectiveness. However, since it is very challenging to correctly implement parallel EAs on the GPU, we propose a set of guidelines to verify the correctness of GPU-based EAs. In order to examine these guidelines, they are employed to verify a GPU-based brain storm optimisation that is also proposed in this thesis.

In conclusion, the comprehensively experimental study is firstly conducted to investigate the impacts of a large population. After that, a SPEO framework based on CPU-only HPC is proposed and is employed to accelerate a time-consuming implementation of EA. Finally, the correctness verification of implementing EAs based on a single GPU is discussed and the SPEO framework is then extended to be deployed based on GPU-enabled HPC.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Research Scope

Evolutionary algorithms (EAs) [1–4] are a broad class of meta-heuristic optimisation algorithms that use mechanisms inspired by biological evolution, such as reproduction, mutation, recombination, and selection. As population-based searching methods, EAs first generate the initial population in the solution space and then iteratively update it by applying the above evolutionary operators until it converges or a certain number of generations is reached. Since EAs ideally do not make any assumption about the underlying fitness landscape, they have been successfully used to solve various real-world optimisation problems [5–10]. However, due to the nature of the stochastic search, EAs usually need a long computational time to find satisfactory solutions [5–8, 10].

Intrinsically, EAs are highly parallel owing to the data-parallel algorithmic structure in which each individual performs genetic operations independent of each other. Thus, this time-consuming process of EAs can be highly accelerated by distributing the considerable computational effort on multiple machines. First introduced in the 1980s, parallel EAs [11–14] deploy the massive computational load onto many machines for parallel processing. Specifically, the existing parallel models of EAs can be divided into population-distributed and dimension-distributed [15–18]. The population-distributed model that includes master-slave [19–22], island [14, 23–25] and cellular model [26–29] distributes the global population to parallel comput-

Table 1.1: The top 10 HPC in TOP500 site at November 2018.

Rank	System	CPU Cores	GPUs	Rpeak (TFlop/s)
1	Summit (United States)	2,282,544	26,136 (Nvidia V100)	187,659.30
2	Sunway TaihuLight (China)	10,649,600	-	125,435.90
3	Sierra (United States)	1,572,480	17,280 (Nvidia V100)	119,193.60
4	Tianhe-2A (China)	4,981,760	-	100,678.70
5	ABCI (Japan)	391,680	4,352 (Nvidia V100)	32,576.60
6	Piz Daint (Switzerland)	361,760	5,320 (Nvidia V100)	25,326.30
7	Titan (United States)	560,640	18,688 (Nvidia K20x)	27,112.50
8	Sequoia (United States)	1,572,864	-	20,132.70
9	Trinity (United States)	979,968	-	43,902.60
10	Cori (United States)	622,336	-	27,880.7

ing facilities and achieves parallelism between individuals. The dimension-distributed model such as coevolution model [30–33] distributes the entire problem to parallel computing facilities and achieves parallelism between elements of solutions. However, limited accelerations were achieved by parallel EAs in the past because scientists can not easily access or afford parallel computing facilities even though they were backward.

With the increasing scale and complexity of real-world optimisation problems, higher computing power has become a necessity. Therefore, high performance computing (HPC), which can scale to extensive computing resources, has become increasingly important in research and engineering fields in recent years. Accordingly, Table 1.1 from the Top500 site (November 2018)[1] indicates that the top 10 HPCs in the world assemble many processors. Furthermore, dynamically provisioned and pay-as-you-go computing resources of HPC are now offered by some cloud computing providers such as Amazon Web Services (AWS), Google Cloud Platform, and Microsoft Azure. Consequently, many compute-intensive tasks, which were impractical in the past, can now be efficiently completed on HPC. Moreover, inspired by the unprecedented computational power of modern HPC, parallel EAs have become popular again and justify a new presentation of the state-of-the-art in the field [34–37]. Real et al. [38] distributed 1,000 individuals onto 250 CPU cores to automatically design a deep neural network structure for image classification. Furthermore, Salimans et al. [39] solved the MuJoCo 3D humanoid task

[1] https://www.top500.org/lists/2018/11/

Table 1.2: The algorithms and their population sizes that win at CEC competitions from 2014 to 2018.

Algorithm	Population	Year	Winner
L-SHADE	2000*	2014	CEC2014 Competition on Single Objective Real-Parameter Numerical Optimization
SPS-L-SHADE-EIG	up to 2046*	2015	CEC2015 Competition on Learning-based Real-Parameter Single Objective Optimization
MVMO	up to 3594	2016	CEC2016 Competition on Learning-based Real-Parameter Single Objective Optimization
			CEC2016 Competition on Real-Parameter Single Objective Computationally expensive Optimization
EBOwithCMAR	6504*	2017	CEC2017 Competition on Single Bound Constrained Real-Parameter Numerical Optimization
L-SHADE44	1800*	2017	CEC2017 Competition on Constrained Real-Parameter Numerical Optimization
HS-ES	683	2018	CEC2018 Competition on Single Bound Constrained Real-Parameter Numerical Optimization
IUDE	500	2018	CEC2018 Competition on Constrained Real-Parameter Numerical Optimization

* The algorithm employs a population reduction strategy that gradually reduces a large population during the period of evolution.

by employing evolution strategies (ES) with a population size of 1,440 as an alternative way of reinforcement learning (RL). The proposed method required 11 hours if only 18 CPU cores were being utilised while being reduced to 10 minutes if 1440 CPU cores were demanded.

In recent years, various dedicated computing facilities which have been optimised for power efficiency, compute-intensive, and throughput-intensive scenarios–have entered mainstream computing. Therefore, general-purpose computing on graphics processing units (GPGPU) is one of the most important heterogeneous solutions. The GPU, which was designed for addressing highly computational graphics tasks since its inception, has many computational cores and can provide massive parallelism at a reasonable price. Moreover, the high performance of floating-point arithmetic and memory operations on GPUs makes them particularly well-suited to several similar scientific and engineering workloads that occupy CPUs. Inspired by the incredible computing capability, many HPCs evolve from traditional clusters of homogeneous nodes (CPU-only) to clusters of heterogeneous nodes (CPU + GPU) (a half of top 10 HPCs

at Table 1.1 are GPU-enabled). Apart from the powerful computing capability, GPU-enabled HPC also has better cost-effectiveness (price/performance ratio) than traditional CPU-only HPC. As a result, GPU-enabled HPC has become an ideal platform for scientific computing and has remarkably accelerated many existing EAs that were sequentially implemented on CPU [40–45].

1.2 Motivations

The increasingly complex and large-scale problems bring the rapidly rising solution spaces and quickly exceed the searching capabilities of traditional EAs. As a result, the approach of using EAs to address these difficult problems is currently attracting significant attention [5–10]. An interesting phenomenon observed in recent works is that many state-of-the-art EAs have started to employ a population that is significantly larger than the size suggested by classic EAs [46–48]. For example, many winners at the famous CEC competitions since 2014 have employed a large population (up to 6504) to achieve satisfactory solutions (see Table 1.2) and many real-world problems [49–53] also employ a large population to achieve satisfactory solutions.

Although large populations have been widely employed, researchers in the domain of EAs have not reached a consensus on the benefits of this approach. Two opposing theoretical views exist, and solid numerical evidence is still lacking. Specifically, some works [54–59] have theoretically proven that a large population can eventually improve the solution quality of problems with complex landscapes if sufficient fitness evaluations (FEs) are provided. Conversely, other studies [60–62] have indicated that a large population does nothing to improve the solution quality. Since most theoretical findings are based on problems and algorithms that can be represented by simple mathematical models (e.g., discrete problems and simple EAs without crossover), it is necessary to examine both opinions by numerical experiments in practice.

On the other hand, EAs with a large population require a significantly long computational time. Although researchers do not yet agree on the benefits of a large population, it is a common view [58, 59, 61] that EAs with a large population usually require significantly large FEs. Therefore, designing parallel EAs with a large population based on modern powerful

HPC facilities may make it possible to achieve numerous FEs in a short time. Since the use of parallel EAs based on HPC is not a new topic in the field of EAs, this thesis mainly focuses on the following aspects that have not been perfectly addressed yet.

Firstly, the scalability is essential in parallel computing and determines whether parallel EAs can efficiently utilise extensive computing resources. As the optimisation problems are increasingly complicated and time-consuming, efficiently utilising more computing resources to employ an even larger population or to achieve better speedup is necessary. However, it is not an easy task because communication and synchronisation costs increase rapidly with the use of more computing resources. To avoid significant deterioration of computational efficiency, most existing parallel EAs [63–66] are designed to use only a small number of CPU cores efficiently. Although a few studies [35, 37, 67–70] have achieved scalability by using many devices, these approaches have had to employ sparse and light communication schemes, which unavoidably sacrifice the solution quality proven by works [66, 71, 72].

Secondly, EAs are intrinsically a class of iterative generate-and-test procedures, which iteratively create new populations based on their previous populations. In such a process, a significantly large number of FEs are employed by a large population, but only a small proportion of candidate solutions are superior and may enter new populations (most candidate solutions will be discarded). In fact, any previously evaluated candidate solutions, whether superior or inferior, may carry some useful information about the search landscape. The landscape can facilitate the search process by producing more superior candidate solutions if the evaluated solutions are properly learnt and used. Therefore, if these evaluated solutions can be reused rather than abandoned as inferior solutions, parallel EAs may provide better solution accuracy for solving difficult real-world problems.

Thirdly, CPU-only HPC with numerous CPU cores is a common platform to accelerate various parallel EAs. Since a single CPU core offers limited computing capability, many CPU cores are necessary to generate and evaluate sufficient candidate solutions in a short time. As a result, the high expense for demanding considerable computing resources is unavoidable. Currently, GPU devices shows the prospect in the field of EA research because even a single GPU can provide similar computing power to a large number of CPU cores with less expense.

For instance, if the work of Liu [69] is executed on the Amazon EC2 CPU instance[2], it will cost USD 696.2 on 16,384 CPU cores for 1 hour; however, executing the same number of floating point operations on the AWS GPU instance would save around 80% of the cost. However, there are also some challenges when utilising GPUs to accelerate parallel EAs:

- Modern GPUs are characterised by inexpensive prices, general-purpose parallel computing infrastructures, and easy-to-use programming models, which have intensified common personal computers (PCs), i.e., desktops and laptops. So far, GPUs have accelerated many time-consuming EAs [73–78] and offered remarkable speedups. However, programming on a single GPU is much more difficult than programming in a serial-oriented order on a single CPU. Parallel programming based on GPU involves many challenges which are not typically encountered in conventional serial-oriented programming. In other words, implementing correct GPU-based EAs is not straightforward, resulting in a high risk that GPU-based EAs may output incorrectly. So far, all existing studies neglect the correctness of GPU-based EAs, making the significant speedups potentially unreliable.

- Although a single GPU is capable of offering considerable computing power, the scalability is still limited because each GPU has a maximal number of GPU processors and cannot offer further more computing power. Therefore, it is ideal to utilise a GPU-enabled HPC that can increase GPU computing power by allocating on-demand GPU devices. However, existing works on GPU-based EAs have been designed for a single GPU [79–82] or a small fixed number of GPUs [83–87] without scalability. As a result, existing parallel EAs are not able to efficiently utilise the unprecedented computing power offered by a large number of GPUs.

1.3 Objectives

Based on the above discussion, several objectives can be summarised as follows.

Objective 1: Comprehensively study the impact of a large population by numerical experiments

[2]https://aws.amazon.com/ec2/pricing/on-demand/

The benefits of a large population have been studied by many theoretical works based on some simplified mathematical models. In this thesis, we tend to examine these theoretical findings by conducting a comprehensive investigation using numerical experiments. The challenge of this work is how to design experiments to comprehensively investigate the impacts of a large population. Specifically, the selected algorithms should range from classic to state-of-the-art or from simple to complex. Moreover, the ideal selected problems should be famous and difficult because supporting theoretical works admit that a large population mainly benefits to solve difficult or complex problems.

Objective 2: Utilise many CPU cores at CPU-only HPC to achieve remarkable improvements on both the computing speed and the solution quality

Currently, CPU-only HPC has become a common parallel computing platform in the scientific computing area. In this thesis, we target at utilising many CPU cores at CPU-only HPC to improve both the computing speed and the solution quality of traditional sequential EAs. The two sub-objectives are as follows:

- We intend to design a scalable parallel evolutionary optimisation (SPEO) framework that can efficiently utilise on-demand computing resources on CPU-only HPC. The proposed framework can be easily employed to significantly accelerate common classic and state-of-the-art EAs by efficiently using a large number of CPU cores. The main challenge of this work is accomplishing the essential scalability, which requires the expertise in parallel computing and is not a straightforward task. Specifically, utilising more devices and maintain adequate information exchange usually significantly increase the communication/synchronisation cost which rapidly occupies the majority of the computational budget.

- We intend to improve the search capability of EAs by learning from the extensive historical information. Due to the significant computing power offered by CPU-only HPC, the proposed framework can create far more candidate solutions than traditional sequential EAs. Therefore, the searching capability of parallel EAs can be highly improved when the extensive historical information carried by these solutions is utilised properly. The main challenge of this work is to find a proper approach to take advantage of this historical

and effectively improves the searching capability of EAs; otherwise, the search process may be misled to an even worse result.

Objective 3: Utilise GPU devices to achieve both reliable solutions and remarkable accelerations

GPU-enabled HPC is a modern parallel computing platform that can offer a significantly large computing power. In this thesis, we target at utilising GPU devices to achieve both reliable solutions and remarkable accelerations. The two sub-objectives are as follows:

- We intend to propose guidelines for researchers to correctly implement parallel EAs based on a single GPU. Such guidelines can assist EA researchers in guaranteeing the correctness of parallel GPU-based EAs. The challenge of this work is the complexity of implementing parallel EAs based on GPU. Specifically, many GPU-inherent factors may bring unexpected outcomes compared to traditional CPU-based EAs and the stochastic searching nature of EAs even aggravates this issue.
- We intend to extend the above SPEO framework from CPU-only HPC to GPU-enabled HPC. The proposed framework can accomplish outstanding scalability to efficiently utilise on-demand GPU devices and requires a lower cost when compared to CPU-based parallel EAs. The two main challenges of this work are as follows.
  - CPU and GPU work cooperatively in a heterogeneous architecture, thus computing tasks can be assigned to either GPU or CPU or a combination of both. Therefore, the mapping choice has a great impact on the computational efficiency of the framework because the CPU and GPU are suitable for different jobs.
  - Compared to traditional CPU-based parallel computing, information sharing between GPU devices is more complex and inefficient. Since each CPU of the existing multi-GPU EAs is in charge of both communication and GPU control, the increasing communication workload rapidly occupies the CPU's computing power, which results in the waiting of launching GPU kernel functions. Therefore, it is challenging but essential to design a proper scheme that can avoid the impacts of extensive communication workload on computing efficiency.

1.4 Contributions

Based on the above objectives and their corresponding challenges, the research contributions are summarised as follows:

Contribution 1: We experimentally study and investigate the impacts of a large population on EAs. Specifically, we select five representative EAs including two state-of-the-art and three classic EAs. We also choose eight complex composition functions from the famous CEC2014 benchmark [88] as test problems. In this work, we examine different population sizes ranging from small (64) to large (4096) using a very large FEs. As an experimental supplement to the theoretical findings, this work confirms the universal benefits of a large population to solve difficult problems. Compared to some complex methods that are carefully designed for specific algorithms, employing a large population is a feasible and simple option to significantly improve the solution quality of various EAs if researchers can access to a large computing power.

Contribution 2: We study utilising many CPU cores at CPU-only HPC to achieve remarkable improvements on the computing speed and the solution quality. We firstly propose a generic scalable parallel evolutionary optimisation (SPEO) framework based on many CPU cores at CPU-only HPC. Based on this framework, we also design and implement a local ensemble surrogate scheme to improve the searching capability of the traditional crowding DE algorithm. The two sub-contributions are described as follows:

- We propose the SPEO on CPU-only HPC ($\text{SPEO}_{\text{HPC}_{\text{cpu}}}$) which achieves outstanding scalability by employing the island model with a buffer-based asynchronous migration strategy. The proposed $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ is designed to be able to execute common sequential implementations of EAs on each CPU core. Based on this framework, EA researchers can significantly accelerate their novel sequential EAs without worrying about the issues related to parallel computing.

- The $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ can produce unprecedented candidate solutions that carry useful historical information and improve the search capability of EAs. In this thesis, we design a local ensemble surrogate crowding DE (LES-CDE) algorithm that significantly improves the

solution quality by learning from the extensive historical information. We also implement LES-CDE based on the $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ and achieve remarkable speedups. It demonstrates that the $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ framework is very ideal to accelerate such data-driven algorithms that require a unprecedented computing budget to learn from historical data.

Contribution 3: We utilise GPU devices to further accelerate parallel EAs and achieve reliable solutions meanwhile. We firstly propose a set of guidelines to assist in correctly implementing GPU-based EAs. Then we extend the SPEO framework onto GPU-enabled HPC so that a correct implementation of parallel EAs based on a single GPU can be simply deployed over multiple GPUs for further speedups. The two sub-contributions are described as follows:

- We firstly figure out some GPU-inherent factors that may bring difficulties when implementing parallel EAs based on a single GPU. Then, we propose a set of guidelines to assist EA researchers in verifying the implementation of GPU-based EAs against these factors. Benefiting from this work, EA researchers can guarantee that parallel GPU-based EAs are correctly implemented. In order to examine the proposed guidelines, we employ them to correctly implement a complex EA called modified brain storm optimisation (MBSO) based on GPU.

- We design an extended SPEO framework ($\text{SPEO}_{\text{HPC}_{\text{gpu}}}$) based on many GPUs at GPU-enabled HPC. In order to efficiently utilise many GPUs and achieve outstanding scalability, the above proposed $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ needs to be adjusted based on the characters of GPU-enabled HPC. Similar to the $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$, the buffer-based island model is still applied by $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ to run EAs independently on many GPUs, which avoids time-consuming synchronisation between GPUs. Furthermore, a dual control mode is additionally introduced to improve the efficiency of CPU-CPU and CPU-GPU communication. Three contributions of this work are listed as follows:

  - The $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ is the first solution that can deploy EAs over a large number of GPUs. It is now possible for EAs to solve some extremely large-scale or time-consuming problems that were impractical in the past.

- The $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ does not limit the GPU-based implementations of EAs, thus researchers who already have implemented a specific EA on a single GPU can easily deploy it over many GPUs without worrying about the essential interactions between GPUs.
- The proposed framework is an ideal option for researchers who have a limited budget because it costs significantly less than CPU-based parallel EAs to achieve similar or better performance.

1.5 Organisation of the Thesis

The remainder of this thesis is organised as follows:

In Chapter 2, the basic concepts of the topics related to this research are introduced. Firstly, The basic knowledge of EAs and three representative EAs including genetic algorithm (GA), differential evolution (DE), particle swarm optimiser (PSO) and brain storm optimisation (BSO) are briefly described. Then we survey the existing works on the population sizing. After that, the basic concepts of parallel EAs and some population topology models are briefly introduced. As a representative population topology model, the island model which works as the foundation of this thesis is described in this chapter. The GPU-based parallel EAs are surveyed then. Finally, we give a brief introduction of the modern HPC facilities and some basic knowledge of GPU computing.

In Chapter 3, we conduct the first comprehensively experimental study to investigate the impacts of a large population on the performance of EAs in terms of the solution quality and computing speed. Specifically, we present that a large population improves the solution quality of two state-of-the-art and three generic EAs on eight difficult benchmark functions. Moreover, we also demonstrate that a large population can bring better parallelism and speedups when implemented in parallel.

In Chapter 4, We propose the SPEO framework on CPU-only HPC ($\text{SPEO}_{\text{HPC}_{\text{cpu}}}$). The proposed framework is implemented based on a standard DE algorithm and its performance is evaluated on eight composition functions of CEC2014 benchmark at the Australian National

Computational Infrastructure (NCI) platform using up to 512 CPU cores. Experimental results demonstrate that $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ is very scalable because approximately linear speedups are achieved. The results also present that $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ not only increases the computational efficiency but also improves the solution quality when compared to a state-of-the-art parallel EA.

In Chapter 5, we proposed the LES-CDE which can use historical search information to improve the searching capability. Specifically, we design an ensemble of several neighbouring local models that are trained by historical information to guide the generation of promising trial vectors. In this work, an online sequential extreme learning machine (OS-ELM) is used to construct and update these models in an online manner. After that, we implement the LES-CDE in parallel based on the $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ framework. The numerical results demonstrate that sequential LES-CDE can achieve significantly better solutions than the original CDE regardless that it requires a very long computing time to train the model. Benefiting from the $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ framework, the parallel LES-CDE can be remarkably accelerated and only requires a very short computing time to achieve extensive FEs. Finally, we provide a recommendation on configuring the chunk and volume of online training data for LES-CDE to achieve both satisfactory solution quality and computing speed.

In Chapter 6, a comprehensive procedure for verifying the correctness of GPU-based EAs is proposed to address the difficult but necessary correctness verification problem. An example of migrating the PSO from CPU based coding to the GPU environment is given as an example. In addition, some GPU-inherent issues, which influence the output of GPU-based EAs including the library functions, the numerical precision, and the race condition, are examined one by one. To cope with the issues mentioned above, a set of guidelines are proposed to verify the correctness of the GPU-based EAs. Finally, we present a working example that applies these guidelines to correctly implement the GPU-based MSBO.

In Chapter 7, we propose the SPEO framework on GPU-enabled HPC ($\text{SPEO}_{\text{HPC}_{\text{gpu}}}$) that works efficiently with many GPUs. The $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ introduces a dual control mode to improve the scalability when an increasing number of GPUs are utilised. The performance of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ is evaluated on eight composition functions of CEC2014 benchmark at NCI using

up to 64 GPUs. Experimental results demonstrate that $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ scales well by obtaining linear speedups and achieves 3,000x speedups compared to its sequential counterpart. Results also demonstrate that the $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ outperforms $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ and a state-of-the-art CPU-based parallel EA with the same computational budget of USD 1, 10, 50 and 100.

Chapter 8 concludes the thesis with some recommended further research directions.

Chapter 2

Background and Literature Review

In this chapter, some concepts of the topics related to this research are introduced. Firstly, The basic knowledge of EAs and three representative EAs including GA, DE and PSO are briefly described. Then we survey the existing works on the population sizing. After that, the basic concepts of parallel EAs and some population topology models are briefly introduced. We detailedly describe the island model which is a representative population topology model and works as the foundation of this thesis. The GPU-based parallel EAs are surveyed then. Finally, we give a brief introduction of the modern HPC facilities and some basic knowledge of GPU computing.

2.1 Evolutionary Algorithms (EAs)

2.1.1 Overview

EAs [1–4] are a broad class of meta-heuristic optimisation algorithms which use mechanisms inspired by biological evolution, such as reproduction, mutation, recombination, and selection. As population-based searching methods, EAs firstly generate the initial population in the solution space and then iteratively update it by applying above evolutionary operators until it converges or a certain number of generations are reached. Since EAs ideally do not make any assumption about the underlying fitness landscape, they are successfully used to solve various real-world optimisation problems [5–10]. DE [47], PSO [89] and GA [90] are some well-known

representatives.

2.1.2 Representative EAs

2.1.2.1 Differential Evolution

The DE algorithm [91–97] is a simple yet powerful population-based stochastic search technique, which is an efficient and effective global optimiser in the continuous search domain. There are also many popular DE variants [98–101] that offer excellent searching capability. In the D-dimensional search space, DE evolves a population of NP individual vectors $\mathbf{X}_G = \{\mathbf{x}_{1,G}, \cdots, \mathbf{x}_{NP,G}\}$, where $\mathbf{x}_{i,G} = \{x^1_{i,G}, \cdots, x^D_{i,G}\}$, $i = 1 \cdots NP$ in quest of globally optimal solutions. The initial population at generation $G = 0$ is randomly generated in the search space. From a certain generation to the next, DE employs several operators, i.e. mutation and crossover, to produce NP trial vectors. Then the trial vectors are evaluated and the target vectors are updated by the selection operator.

2.1.2.1.1 Mutation

At generation G, a mutant vector $\mathbf{v}_{i,G} = \{v^1_{i,G}, \cdots, v^D_{i,G}\}$, $i = 1 \cdots NP$ is produced from a target vector $\mathbf{x}_{i,G}$ by mutation operation. The five frequently used strategies are as follows:

- **DE/rand/1:** $\mathbf{v}_{i,G} = \mathbf{x}_{r_1,G} + F \cdot (\mathbf{x}_{r_2,G} - \mathbf{x}_{r_3,G})$
- **DE/best/1:** $\mathbf{v}_{i,G} = \mathbf{x}_{best,G} + F \cdot (\mathbf{x}_{r_1,G} - \mathbf{x}_{r_2,G})$
- **DE/rand-to-best/1:** $\mathbf{v}_{i,G} = \mathbf{x}_{r_1,G} + F \cdot (\mathbf{x}_{best,G} - \mathbf{x}_{r_1,G}) + F \cdot (\mathbf{x}_{r_2,G} - \mathbf{x}_{r_3,G})$
- **DE/current-to-best/1:** $\mathbf{v}_{i,G} = \mathbf{x}_{i,G} + F \cdot (\mathbf{x}_{best,G} - \mathbf{x}_{i,G}) + F \cdot (\mathbf{x}_{r_1,G} - \mathbf{x}_{r_2,G})$
- **DE/rand/2:** $\mathbf{v}_{i,G} = \mathbf{x}_{r_1,G} + F \cdot (\mathbf{x}_{r_2,G} - \mathbf{x}_{r_3,G}) + F \cdot (\mathbf{x}_{r_4,G} - \mathbf{x}_{r_5,G})$
- **DE/best/2:** $\mathbf{v}_{i,G} = \mathbf{x}_{best,G} + F \cdot (\mathbf{x}_{r_1,G} - \mathbf{x}_{r_2,G}) + F \cdot (\mathbf{x}_{r_3,G} - \mathbf{x}_{r_4,G})$

Where the distinct integer indices r_1, r_2, r_3, r_4 and r_5 are randomly selected in the range $[1, NP]$ and are also different from the target vector's index i. The scaling factor is a positive F control

parameter for scaling the difference vector. $\mathbf{x}_{best,G}$ is the best member in the population at generation G.

2.1.2.1.2 Crossover

Crossover operator generates the i-th trial vector $\mathbf{u}_{i,G} = \{u^1_{i,G}, \cdots, u^D_{i,G}\}$ from the mutant vector $\mathbf{v}_{i,G}$ and its corresponding target vector $\mathbf{x}_{i,G}$

$$u^d_{i,G} = \begin{cases} v^d_{i,G} & if\ (rand_j\,[0,1] \leq CR)\ or\ (j = j_{rand}) \\ x^d_{i,G} & otherwise \end{cases} \qquad d = 1, 2, \cdots D$$

Where $CR \in [0, 1)$ is set by users as the crossover rate parameter and j_{rand} is a random integer in the range $[1, NP]$ to ensure that the trial vector $\mathbf{u}_{i,G}$ is different from its corresponding target vector $\mathbf{x}_{i,G}$

2.1.2.1.3 Selection

After generating a trial vector, the constraint checking will be applied to ensure that this newly generated trial vector is a feasible candidate solution. We will apply random re-initialisation or some other schemes to repair an infeasible trial vector to make the repaired one to satisfy the constraints. Then, a set of generated trial vectors $\mathbf{U}_G = \{\mathbf{u}_{1,G}, \cdots, \mathbf{u}_{NP,G}\}$ are evaluated in terms of their objective function values denoted by $F(\mathbf{U}_G) = \{f(\mathbf{u}_{1,G}), \ldots, f(\mathbf{u}_{NP,G})\}$, Each trial vector $\mathbf{u}_{i,G}$ will compare its function value $f(\mathbf{u}_{i,G})$, with the function value $f(\mathbf{x}_{i,G})$ of its corresponding target vector $\mathbf{x}_{i,G}$. If the trial vector performs better than the target vector, it will replace the target vector and enter the next population. Otherwise, the target vector will be retained in the next population. This operation is described as follows:

$$\mathbf{x}_{i,G+1} = \begin{cases} \mathbf{u}_{i,G} & f\,(\mathbf{u}_{i,G}) < f\,(\mathbf{x}_{i,G}) \\ \mathbf{x}_{i,G} & otherwise \end{cases}$$

2.1.2.2 Particle Swarm optimisation

PSO [89, 102–111] is a family of swarm intelligence algorithms inspired by the bird's creative problem-solving process, which has achieved successes in various applications. It does not use

evolution operators such as crossover and mutation; instead, the particle in the swarm adapts its search behaviour by learning from knowledge not only obtained from its own but also from other particle in the whole swarm. These phenomena are studied by mathematical models. In PSO, each particle has a position and a velocity which are adapted based on the best knowledge of each individual and the global best knowledge from the entire swarm as follows [89]:

$$v_{i,G}^{d} \leftarrow v_{i,G}^{d} + c_1 * rand_1 * (pbest_i^d - x_{i,G}^{d}) + c_2 * rand_2 * (gbest^d - x_{i,G}^{d})$$

$$x_{i,G}^{d} \leftarrow x_{i,G}^{d} + v_{i,G}^{d}$$

where $\mathbf{x}_{i,G} = (x_{i,G}^1, x_{i,G}^2, \ldots, x_{i,G}^D)$ is the position of the i-th particle; $\mathbf{v}_{i,G} = (v_{i,G}^1, v_{i,G}^2, \ldots, v_{i,G}^D)$ represents velocity of particle i. $\mathbf{pbest}_i = (pbest_i^1, pbest_i^2, \ldots, pbest_i^D)$ is the best position that has been found by the particle i ; and $\mathbf{gbest} = (gbest^1, gbest^2, \ldots, gbest^D)$ is the best position discovered by all particles in the whole swarm. c_1 and c_2 are the two parameters that control the weight how much each particle learns to *pbest* and *gbest*, respectively. $rand_1$ and $rand_2$ are uniform random numbers ranging $[0, 1]$ to avoid that all particles have the same behaviour. If the absolute speed of any particle exceeds a maximum value v_{max}, the velocity of that dimension is assigned to $sign(|v_i^d|)v_{max}^d$.

2.1.2.3 Genetic Algorithm

GA [112–116] is a metaheuristic technique that simulates the natural selection and genetic mechanism of Darwin's theory of natural evolution. GA initialises and evolves a population with NP chromosomes $\mathbf{X}_G = \{\mathbf{x}_{1,G}, \cdots, \mathbf{x}_{NP,G}\}$ at generation G, where $\mathbf{x}_{i,G} = \{x_{i,G}^1, \cdots, x_{i,G}^D\}$, $i = 1 \cdots NP$. In each generation G, based on a certain selection strategy, several promising parents are selected from the current population to generate offspring based on genetic operations including crossover and mutation. The best NP chromosomes are selected among current population and all newly generated offspring as the new population for generation $G + 1$.

2.1.2.3.1 Selection

GA is usually designed to select the solutions with better quality to inspire the entire population evolve towards a better quality. Specifically, the chromosomes with high fitness values have a

larger chance of being selected. One common selection is roulette wheel selection [117] which is also known as fitness proportionate selection. In roulette wheel selection, each chromosome is selected according to the probability which is calculated as follows:

$$p_i = \frac{f_i}{\sum_{j=1}^{NP} f_j}$$

where f_i is the fitness value of chromosome i. Accordingly, chromosome with a higher fitness value will be less likely to be eliminated.

2.1.2.3.2 Crossover and mutation

The crossover operation, also known as recombination, combines parents to creates new offspring. Here we review and list some popular crossover schemes which are available for binary representation and real parameter optimisation respectively.

- Single point crossover [118] first randomly select a certain gene of parents' chromosomes a crossover point. Two parents swap their chromosome based on this point. Specifically, the genes at the left side of crossover point is kept the same as the original parent and the genes at right side of chromosome are swapped between two parents.

- Linear crossover [119] generates three candidate offspring based on two randomly selected parents $\mathbf{x}_{i,G}$ and $\mathbf{x}_{j,G}$ as follows, where i and j are two different indices selected from $[1, NP]$. Linear crossover selects the two best candidate offspring as the new offspring.

$$\mathbf{x}'_1 = 0.5 * \mathbf{x}_{i,G} + 0.5 * \mathbf{x}_{j,G}$$

$$\mathbf{x}'_2 = 1.5 * \mathbf{x}_{i,G} - 0.5 * \mathbf{x}_{j,G}$$

$$\mathbf{x}'_3 = 1.5 * \mathbf{x}_{j,G} - 0.5 * \mathbf{x}_{i,G}$$

The mutation operation changes some genes of chromosome to explore in searching space where may not be searched before. to improve the diversity of population. Each gene is selected to be mutated based on a regulated mutation rate MR and the value of this gene randomly changes if it is selected. For example, if the k-th gene is selected as the mutation point, the original chromosome $\mathbf{x}_{i,G} = \{x^1_{i,G}, \ldots, x^k_{i,G}, \ldots, x^D_{i,G}\}$ changes to $\{x^1_{i,G}, \ldots, \hat{x}^k_{i,G}, \ldots, x^D_{i,G}\}$, where $\hat{x}^k_{i,G}$ is randomly generated in the searching space.

2.1.3 Brain Storm Optimisation

The brain storm optimization [120] algorithm works as a kind of search space reduction algorithm, the N ideas are grouped into M clusters eventually and the best idea in each cluster is denoted as the representation. Then, the new idea is generated based on selected ideas from one or two clusters. The two important operators are as follows.

- Convergent operator: the convergent operator groups all the current ideas into several clusters based on their difference in solution space. k-means clustering algorithm is applied in this operator, and the best idea in each cluster is set as the centre of this group when k-means finishes.

- Divergent operator: the new ideas are created by current ideas which are randomly selected from one or two clusters and added by a random noise. In Shi's work, a uniform random number from [0,1] is used in random selection and Gaussian random noise $N(0, 1)$ is used for random noise addition.

Although these implementations can make the BSO algorithm able to work, it would be faced with the time-consuming problem when solve high dimension problems. However, considering that an accurate clustering is not necessary for BSO, various variants are designed to reduce the computation time on convergent operator. Modified BSO (MBSO) [121] that is a promising variant shares the same architecture with original BSO.

- Convergent operator: A simple grouping method (SGM) is introduced to group all the current ideas instead of k-means. The new method separates all the ideas based on randomly selected cluster centres, which avoids iteration of distance calculation.

- Divergent operator: In stead of using a fixed logarithmic sigmoid transfer function based on the generation and without feedback information from the search process, MBSO introduces idea difference strategy (IDS) to generate new ideas. IDS taking account of the difference of current ideas when creating new ideas.

2.2 Population Sizes in EAs

As population-based optimisation algorithms, EAs search for good solutions by updating the population and converging to the optima by generations. Specifically, the initial population is randomly generated in the solution space at the beginning of process; then the new population is iteratively reproduced by applying evolutionary operators (e.g. crossover and mutation) until it converges or a certain number of generations are reached. Consequently, the characters of population impact on how the quality of the solutions is and how long it takes to find them, for example, a smaller population may converge more efficiently [122–124] but a larger population may have a greater global search capability [54–59, 61]. In this section, we provide a brief survey on two views 1) a large population can improve the solution quality, especially for complex problems 2) a large population is unhelpful and a small population is sufficient.

2.2.1 EAs with a Large Population

2.2.1.1 Theoretical Analysis

Due to the significant impacts of population size on performance of EAs, the ideal size of a population is studied in many theoretical works. Goldberg [125] studied the optimisation accuracy achieved by GA with a large-enough population regardless of the computing budget. They stated that GA makes many errors of decision and buffeted by the vagaries of chance when a small population size is employed; while GA becomes reliably discriminate between good and bad building blocks if a large population is used. Summarily, if enough computing resources are provided, GA with a large population can be more reliable than a small population to find the global optima. In 2013, to figure out the relationship between optimisation accuracy, reliability and population size, Goldberg [126] further theoretically analysed the impacts of population size on GA and claimed that stochastic effects on performance by a small population can be improved by a sufficient large population size. It is affirmed the benefits of a large population by providing an equation of a conservative population bound. The equation indicates that a increasing population size is required when 1) probability of error is decreased, 2) noise increases, 3) cardinality of the schema increases and 4) signal difference decreases.Additionally,

he also analysed the multiple building block cases and indicated that the population sizes must increase exponentially in number of building blocks to guarantee the converge to the global optima if a complex problem has multiple difficult building blocks. Summary, this work proved that GA requires exponentially large population sizes to ensure convergence to good solutions reliably and + that GA with a sufficiently large population size was able to find one of the 32 global solutions (among the $5.2 * 10^6$ local optima).

Jansen et al. [127] pointed out that for a simple problem, a small population can also have a high probability of successfully finding the global optima; while for a complex problem, a large population is required to find global optima with high probability. Specifically, it is pointed out that an increasingly large population can improve the quality of solutions by avoiding falling into local optima when solving a difficult and complex problem. They used empirical analyses to verify their findings and presented that (1+λ) EA can always find the global optima even with a small λ when the dimension n of complex problem SUFSAMP is small ($n < 24$). It is because the landscape of SUFSAMP with a small n is not complex enough. When the dimension becomes large ($24 < n < 90$), the solution quality of EA with an increasing $\lambda \in [1, n]$ can be improved significantly. Authors also inferred that sufficient function evaluations are very necessary for large a λ because they observed that a large population size may not further improve the solutions if function evaluations are insufficient. Summary, a large λ is necessary and beneficial for EAs when solving complex problems.

Thomas et al. [58] presented that smaller populations yield better results at the beginning of the process but are outperformed by the larger populations eventually. Witt [59] figured out that an EA with a large population size outperforms the same EA with a small population because a large population can provide sufficient exploration and avoid trapping into local optima.

Rowe and Sudholt [128] extended the theory of non-elitist evolutionary algorithms (EAs) by considering the offspring population size in the (1,λ) EA. They pointed that the (1,λ) EA needs exponential time on every function that has only one global optimum if population size is small. They also studied arbitrary unimodal functions and investigated the threshold for offspring population size. They found that this threshold is preferred to be shifted towards a

larger value. Finally, they conducted an experimental study and demonstrated that a large λ requires less generations to solve OneMax, LeadingOnes, and Ridge problems.

Gieben and Witt [129] obtained the theoretical results by a careful study of order statistics of the binomial distribution and variable drift theorems for upper and lower bounds. Based on OneMax problem, they figured out that a large population makes the algorithm more robust with respect to the choice of the mutation probability.

Qi and Palmieri [130] established fundamental theoretical properties of population size for GA in continuous space. They let the population size go to infinity and deriving the consequent limiting behavior of selection, mutation, and crossover. They pointed put that population members tend to cover the entire solution space continuously as the population size gets large. After that, they showed that the probability density function (PDF) of the population will be narrowly concentrated around the global maximum after sufficiently long computing time.

Apart from these works which are regardless of considerations of computational cost, some works [58, 59, 61] pointed out that sufficient fitness evaluations are necessary to converge a large population to an optima.

2.2.1.2 Empirical Analysis

Although theoretical analyses claim that a large population benefits to the exploration of EA with sufficient computing budget, it is necessary to employ experiments to examine whether EAs with a large population is practical to solve optimisation problems.

Costa et al. [131] investigated an empirical comparative study of EAs with the different population size (50, 100, 512 and 1000). It is observed that GA perform best with a population size 512 on on several problems. According to authors' view, population size 1000 performs worse than 512 due to the shortage of function evaluations.

Hu and Banzhaf [132] tested the Genetic Programming with different population sizes (200, 2000 and 20000). Results indicated that a larger population is better at searching and maintaining a high quality of solutions than smaller population. However, since the authors measured the evolutionary speed by the number of generations which results in more fitness

evaluations are a large population to reach the same generation as a small population.

Tung and Yu [133] investigated the performances and behaviors of convergence in optimal mixing evolutionary algorithms and examined the performance of population bound equation proposed by Goldberg [125, 126]. Experiments indicated that the success rate increases from 0 to 1 with an increasing population size from 50 to 300. They also studied the convergence time with one mask of size 5 of OMEA with different population sizes and observed that the time increases when population increase from 1200 to 10000 but keeps stable from 10000 to 204800. It is very interesting to see when solving problems with $l = 100$ with masks of size 5, the convergence time of population 100000 is even shorter than 200 which is also match some theoretical findings by [125, 126].

Friedrich et al. [134] analysed and compared various diversity-preserving mechanisms for global exploration including crowding, fitness sharing and so on. A simple bimodal test function for 30 dimension and rigorous runtime analyses are employed in this study and the population size increases from 2 to 1024. Results show that a larger population always has a higher success rate than a small rate for all diversity-preserving mechanisms.

Hansen and Kern [53] studied the impacts of the population size λ on the performance of CMA-ES. They found that CMA-ES, which is designed for a small population size, can be remarkably improved with a large population size. In this work, CMA-ES with population size from 5 to 1000 are compared on eight numeric test functions. Results shows that a large population can always achieve higher success rate compared to a small population.

Belkhir et al. [52] also investigated the effects of a large population on performance of CMA-ES and employed a self-CMA-ES to better utilise a large population size. They tested the CMA-ES on BBOB benchmark which containing 24 functions and set the λ from 10 to 1000. Results indicated that the self-CMA-ES with a large population outperform a CMA-ES with default parameters.

De Jong and Spears [135] presented some theoretical and empirical results on the interacting roles of population size and crossover in genetic algorithms. They firstly proved that the crossover productivity effects are much less dramatic if a large population is used. They tested population sizes ranging from 20 to 1000 and demonstrated that a larger population results in

better solutions, although the GA must be run for a greater number of generations.

Zhang et al. [136] examined effects of genetic fluctuations on the performance of GA calculations. They considered the roles of mutation by using the stochastic schema theory within the framework of the Wright-Fisher model of Markov processes. The success probability of obtaining the optimum solution was investigated experimentally and theoretically. They conducted the experiments using population from 10 to 500 and noticed that the numerical calculations approaches the theoretical result when the population is large. On the contrary, there will be little change for the convergence of average fitness the small population.

Mallipeddi and Suganthan [137] investigated the effect of population size on the quality of solutions and the computational effort required by the DE Algorithm. The experiments are conducted on various population sizes ranging from $2D$ to $10D$, where D is problem dimension (10 and 30). From these experiments, they observed that a large population with a strategy having good exploration capacity reduces the probability of premature convergence and stagnation effects, but the convergence speed can be slower.

Oda et al. [138] dealt with the effect of changes in population size and number of generations for node placement problem in Wireless Mesh Networks (WMNs). They considered two population sizes 8 and 512 and for every population size the number of generation are 200 and 20,000. Thus, the increase of the population size results in better performance behaviour. However, when the number of generation is also increased, the computation time is increased.

Sarkers and Kazi [139] investigated the effect of population sizes on the quality of solutions to be obtained, the computational time to he required and the size of search spaces of the problems under consideration. They selected a two-stage transportation problem as a test case and also used a well-known conventional optimization technique to compare the solutions. The experimental results are performed on population sizes ranging from 50 to 2000. The results state that the population sizes may need to increase for improving the quality of solutions for a two stage transportation problem, and it may need further increasing for higher dimensional models. In summary, they concluded that the quality of solutions is highly dependent on the population size specifically when the search space is larger.

Piotrowski [140] briefly reviewed the opinions regarding DE population size setting and

verified the impact of the population size on the performance of DE algorithms. Ten DE algorithms with fixed population size, each with at least five different population size settings, and four DE algorithms with flexible population size are tested on CEC2005 benchmarks and CEC2011 real-world problems. According to the numerical results, a large population size is useful when the problem is very hard and multimodal. Moreover, too low population sizes may diminish the number of available moves and prevent convergence within the specified number of function calls, even in case of unimodal problems. Moreover, they also stated that some variants of DE with flexible population size do not outperform the variants with fixed population size, if the fixed population is configured properly.

Zhan et al. [141] studied the population size how to impact on convergence rate, convergence time and global search capability of the genetic algorithm for the typical benchmark functions. According to their work, the increase of population size will reduce the evolution generation numbers if the total fitness evaluations are fixed and the global search capability still enhance.

Hernhdez-Aguirre et al. [142] used the Probably Approximately Correct (PAC) framework to derive the size of a GA population. Their experiments used population sizes ranging from 66 to 2518 and demonstrated that small populations converge quickly without finding the solution. On the contrary, large populations have greater chances to find the solution but the consequences are paid in long processing time.

Belmont-Moreno [143] selected a particular set of test problems make an empirical study of a standard algorithm observing general trends when used with the set of test problems. In this work, the behavior of two particular parameters in GA is analyzed. In the experiments, the various population size that are up to 3000 are tested with various total numbers of fitness evaluations that are up to 300,000. It can be observed that if a big number of evaluations is necessary, the optimum population size moves into a bigger number. If a small or moderate number of evaluations are used, there exists a favored population size.

Dong and Yao [144] studied the impacts of population size on performance of Estimation of distribution algorithm (EDA). They figured out that classical EDAs with a small population that use maximum likelihood to estimate Gaussian usually fail because the exploring effectiveness will be fast deteriorating and premature convergence will arise. The performance

of different population sizes ranging from 100 to 2000 are compared by experiments on several test functions. Results shows that sufficiently large population sizes assist the maximum likelihood estimates to be precise and reliable, while a insufficient population performs poor and unstably. Summary, authors concluded that if more efforts are devoted into tunning and algorithm design, the population size can be reduced to save computing budget; however, it is very difficult for EDA to solve difficult problems with with a too small population.

Currently, benefiting from the modern parallel / distributed computing facilities (e.g., GPGPU, cloud computing and HPC), the impacts of population are also studied based on these parallel computing platforms. Folino et al. [67] proposed a scalable cellular parallel GP and employ population sizes ranging from 800 to 6400 on up to 64 computing nodes. Results shows that parallel GP with a larger population converges faster and achieve better solutions than a smaller population when executing on parallel computing platform. Tatsukawa et al. [145] used a many-objective EA designed for massive parallelisation (CHEETAH) on the K supercomputer. They compared the performances of different population sizes from 100 to 1,000,000 on test problems DTLZ1, DTLZ2, DTLZ3 and DTLZ4 at up to 4000 cores on super computer. Results showed that a larger population can achieve significantly better solutions than a smaller population.

2.2.1.3 Applications

So far, there are many applications which employ EAs with a large population to solve difficult optimisation problems especially for some complex real-world problems such as protein structure prediction and neural network design.

Li et al. [50] proposed a novel graph-based EDA and employ reinforcement learning (RL) to enhance the performance in terms of fitness values, search speed, and reliability. The proposed EDA employed a large population 1800 and achieve significantly better solutions compared to genetic network programming (GNP) or GP with a relatively small population size of 300. Valdez et al. [49] proposed a Boltzmann based EDA and showed that different problems require different population size. In the experiments, population sizes are set ranging from 200 to 2400 for 2 to 80 variables. Hans-Georg and Sendhoff. [51] proposed two evolutionary strategies (ES)

for the optimisation of problems with actuator noise as encountered in robust optimisation. they stated that a large population does improve the final solution quality if strong noise exists and examined this idea by employing a (320,800) ES on Sphere function.

Since many works indicated that a large population may result in very slow computing speed, parallel computing becomes increasingly popular on applying a large population. Thus, effective utilizing a large population and processing a large number of fitness evaluations in a reasonable time is no longer impossible. Consequently, various works have utilised parallel EAs with a large population to resolve complicated real-world applications which are far too difficult for a small population before. Real et al. [38] applied parallel EA with a 1000 population on 250 CPU cores for 256 hours to automatically design a neural network for image classification and achieved excellent accuracy compared to manually designed ones. Salimans et al. [39] explored the use of ES as an alternative approach to solve RL problems. In this work, they had been able to use EA with with a 1440 population to solve the MuJoCo 3D humanoid task within 10 minutes on 1440 CPU cores. Roy et al. [146] proposed a new distributed architecture for GAs based on distributed storage of the individuals in a persistent pool. In the experiments, up to 32 threads are used to evolve up to 200 individuals on each thread; in a word, up to total 6400 individuals are tested. Desell et al. [147] discussed different strategies for computing EAs on distributed environments. In particular, sequential strategies which require synchronization between successive populations are compared to asynchronous strategies that do not have explicit dependencies. In the experiments, a population up to 1000 are used to solve Ackley, Rastrigin and Rosenbrock problems. Luque et al. [148] implemented an asynchronous parallel cellular GA for combinatorial optimisation. It used up to 8 processors to accelerate 800 individuals. Li and Wang [69] proposed a scalable parallel genetic algorithm which can process an extremely large population size up to 1,638,400 for the Generalized Assignment Problem (GAP) on HPC with up to 16,384 CPU cores.

2.2.2 EAs with a Small Population

There are also many works support the benefits of utilsing a small population. Most early works on classical EAs applied a relatively small population, for example, the population size

was suggested at 100 for standard GA [46], a population ranging from 5 to 100 is suggested by the DE [47], and 20 for PSO [48].

Chen et al. [60] conducted theoretical study and analyzed the role of population further in EAs and showed rigorously that large populations may not always be useful. They also discussed the conditions under which large populations can be harmful. Specifically, their study was based on the TrapZeros problem and the (N + N) EA without the crossover operation. According to their research, a large population may not be useful and even becomes harmful when a problem has an attraction basin leading to some local optimum, and the individuals at this basin are with relatively high fitness than most individuals.

Cabrera and Coello [149] presented a multi-objective EA (MOEA) based on PSO algorithm which is characterized for using a very small population size. The experiments are conducted using a small population that only has 5 individuals. This small population size combined with a good mechanism to preserve diversity allows them to produce reasonably good approximations of the Pareto front of several test problems of moderate dimensionality (up to 30 decision variables), while performing only 3,000 objective function evaluations.

Many EDAs use a large population size to better represent the landscape of problems. Hong et al. [150] firstly illustrated why EDA does not work well under small population size. However, they then proposed a novel approach termed as over-selection to boost EDA under small population size. Experiments were conducted on several benchmark problems using a relatively small population size 100 for the proposed approach and were compared it to the performance of uni-variate marginal distribution algorithm (UMDA) under four different sizes (M = 15,M = 50,M = 100,M = 500). The results demonstrated that the over-selection with a small population is often able to achieve a better solution without significantly increasing its time consumption when compared with the original version of EDA with a large population.

Ashlock [151] compared the use of various population sizes for three genetic programming problems: 4-parity using parse trees, Tartarus using ISAc lists, and several versions of plus-onerecall-store (PORS) using parse trees. According to experiments based on population sizes that range from 4 to 1000, the best results were obtained by the smallest population for all problems.

Mao and Li [152] employed the MPGA to retrieve the dust particles size distribution using AOT data taken by a sun photometer CE-318. The results showed that the MPGA presents better properties when compared with the SGA; specifically, it requires smaller population size (population 30 for MPGA and 200 for SGA) and fewer generation numbers (generation time 25 for MPGA and 50 for SGA) to achieve smaller inversion errors to retrieve the aerosol size distribution.

Kok et al. [153] conducted on the performance of GA at various selection schemes and population sizes (from 10 to 50). Results showed that large population sizes do not contribute in improving the performance of GA; namely, increasing population size does not considerably improve the convergence speed at least in path planning.

Alander [154] investigated the optimum population size for GA and conducted the test on a sequential machine. Population sizes are tested on up to 120 and the results claimed that a large population is less appealing if fast convergence or great divergence is aimed at.

Haupt [155] conducted experiments to determine the optimum population size and mutation rate for a simple real GA. The experiments were based on various population size ranging from 4 to 128. The results of this investigation showed that a small population size and relatively large mutation rate is far superior to the large population sizes and low mutation rates that is used by most of the papers presented in the electromagnetics community and by the GA community.

Mora-Melia et al. [156] analyzed the optimal size of EA in designing water distribution networks. Experiments demonstrated that large initial population sizes are not more efficient than small populations in finding the best solution. Specifically, for $P > 50$ on Hanoi and GoYang network, the efficiency decreases as the size of population increase; for New York network, there is no significant differences between population sizes are appreciated for $P > 75$.

Nodehi [157] used a novel functional sized population Quantum Evolutionary Algorithm for fractal image compression. The experiments are conducted on three images that are Lena, Pepper and Baboon using small population sizes raning form 15 to 30. The results turned out satisfactory enough and there is no need to employ a larger population size.

Allia et al. [158] investigated the effect of population sizes from their proposed method

of feature selection on different learning classifier algorithms using Random Forest, Voting, Decision Tree, Support Vector Machine and Stacking. Experiments on Ling-Spam email dataset using small population sizes from 1 to 9 demonstrated that even by using the smallest size of the population, it is still able to produce a good result.

Zhang et al. [159] presented a hybrid PSO approach with small population size (HPSO-SP) for solving the optimal short-term HTUC problem. A small population size 5 is used in the experiments and satisfactory solutions are achieved even such a small population is employed.

2.3 Parallel EAs

2.3.1 Overview

In order to efficiently deploy the computational tasks into parallel computing platforms for speedups, there are two main types of population topology models which are population-distributed and dimension-distributed [15–18]. The population-distributed model that includes master-slave [19–22], island [14, 23–25] and cellular model [26–29] distributes the global population to parallel computing facilities and achieves parallelism between individuals. The dimension-distributed model such as coevolution model [30–33] distributes the entire problem to parallel computing facilities and achieves parallelism between elements of solutions.

- The master-slave model is the most straightforward method to distribute time-consuming computational tasks into parallel computing facilities. Single master only does light tasks such as crossover and mutation, while the fitness evaluation which is usually the most time-consuming part is distributed to many slaves. Each slave only receives a portion of global population from the master and sends them back to the master after the evaluating fitness values of individuals on the salver. Communication only exists between master and slave and slaves work independently.

- The island model is a coarse-grained parallel model of EAs and is conceptually a rather simple enhancement to a standard EA. The island model deploys a single global population into several sub-populations (a.k.a. islands) and exchanges information by migration.

Each island shares the information with its neighbor island as defined in the graph of possible inter-island links commonly referred to as migration topology.

- The cellular model is a fine-grained parallel model of EAs. In cellular model, the global population are divided into individuals and assigned to each machine. The interaction such as crossover between individuals are realised through the communication defined by a network topology. Each individual can only interacts with its neighbors and thus promising individual spreads its information gradually to the entire global population.

- The coevolution model is a dimension-distributed model. Unlike to master-slave, cellular and island model, the coevolution model decomposes the complex problem into simple problems and deploys all dimensions into different machines. The basic approach of cooperative coevolution is to divide a large system into many modules and evolve the modules separately. Then these modules are combined again to form the whole problem.

2.3.2 Island Model

As a coarse-grained parallel model of EAs, the island model is conceptually a rather simple enhancement to a standard EA. The island model can outperform standard EAs and is ideal for parallelisation due to the two following benefits. Firstly, the island model is capable to achieve better solutions than standard EAs with a single population because many islands usually provide abundant searching behaviors and improve the diversity of the global population. Secondly, the island model is ideal to implement on multiple machines because all islands can run genetic operations simultaneously. Compared to the master-salve and the cellular model that require communications or synchronisation at each generation, the island model spends less on non-computational tasks since islands only interact in several generations instead of in each generation.

2.3.2.1 Basic Parameters

Basically, the global population with NP individuals are equally divided into several islands. Island size N_s represents the number of individuals in a single island which is the smallest

unit processed by a single EA. Island number M represents the total number of islands which usually equals with the number of machines.

Another two crucial migrated-related parameters are migration interval I and migration rate R_m which determine how often the migration occurs and how many individuals are migrated, respectively. Some works [25, 160] pointed out that a medium migration interval is preferred for the island model because frequent migrations with a small I causes all islands homogenous and lose global diversity. On the contrary, a rare migration with a large I may lead to insufficient information exchange. Regarding migration rate R_m, it does not impact as much as interval on the solution quality, thus a small migration rate is sufficient according to the work [25].

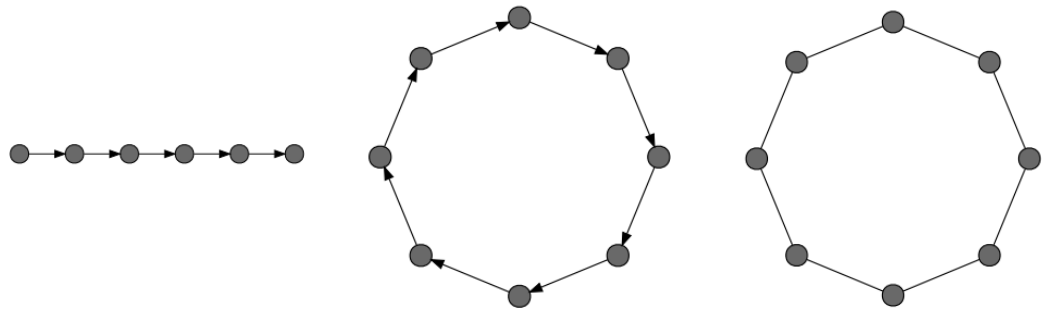
Some works studied dynamically adapting the migration interval and rate for better performance. Whitley et al. [161] configured the interval and migration rate based on the island size when solving linearly separable problems. Specifically, the interval and the migration rate are set 250 and 4% when island size is 50; when the island size increases to 1000, the interval increases to 5000 and the migration rate decreases to 0.5%. Gong and Fukunaga [63] replace the interval parameter with dynamically sending individuals to other islands. The migration occurs unless the best individual in island is updated, by which the algorithm can effectively avoid sending the same or less promising individuals to other islands. Liu and Wang [69] employed different migration intervals for exporting and importing by utilizing a buffer-based asynchronous migration strategy. Zhan et al. [36] proposed an cloud-based island model and replaced the constant interval with a non-linearly increasing probability as follows:

$$p_m = 0.01 + 0.09\frac{exp(\frac{10G}{G_{max}})}{exp(10) - 1} \in [0.01, 1.00]$$

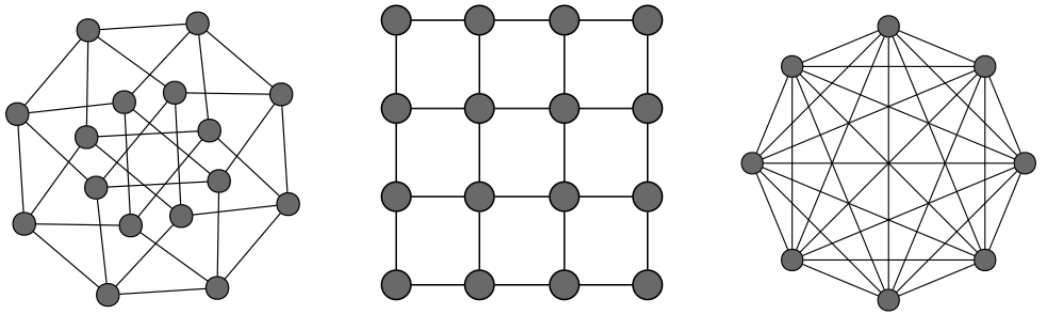
where G is the current generation and G_{max} is the maximal generations. The probability increases from 0.01 at the beginning to 1.00 at the end of algorithm. By this scheme, CloudDE can balance the exploration and exploitation at different stages of the algorithm.

2.3.2.2 Migration Policy

The migration policy determines how the emigrants are selected from the current island and how to insert the immigrants into the recipient island. Basically, migration policy in island



(a) Chain topology (b) Uni-directional ring topology (c) Ring topology



(d) Hypercube topology (e) Lattice topology (f) Fully connected topology

Figure 2.1: Common migration topologies in the island model.

model can be narrowed down to randomness and elitism [162]. For random-based migration policy, one or more individuals in the island are randomly selected or replaced; while elitism select the best individuals as emigrants and replaces the worst ones in the island by immigrants. Araujo and Merelo [24] claimed that elitism-based migration policies increase the selection pressure and can significantly speedup the converge; however, an excessively selection pressure may also lead to converge prematurely. On the other hand, if the island size is small or medium, randomness policy can prevent the "conquest" effect in the recipient island. Zhan et al. [36] employed a randomness policy which directly insert immigrants into the island instead of replacing certain existing island members.

2.3.2.3 Migration Topology

The island model exchanges the information based on the graph of links between islands which can be defined as the migration topology. The migration topology determines the path and speed of spreading useful information among islands [66, 163], specifically, a topology with a smaller diameter usually spreads the information faster [66]. For example, if totally M islands are employed, a fully connected topology can exchange information between any two islands by a single migration; while a chain topology require at least $M-1$ steps from the head to tail island. Figure 2.1 shows some frequently used migration topologies. Rucinski et al. [66] studied and compared the performance of 14 migration topologies including ring, chain, torus, lattice, hypercube, fully connected. Experiments demonstrated that no topology can perform well on all problems and the best option for specific problems or algorithms always depends. However, a dense topologies are usually a safer choice because they perform more stably than sparser topologies. Many works [64, 71] also proved that denser topologies brought better solutions but are also more costly than sparse topologies due to the larger demand of communication.

Besides these static migration topologies, there are also some dynamic migration topologies. Standard dynamic topology [163] does not predefine any connection between islands, instead, each island randomly selects an island as its recipient in each migration. Some dynamic topologies works based on overall quality of islands. Zhan et al. [36] proposed the CloudDE algorithm with a dynamic migration topology which lets some individuals migrate from a relatively poorly-performed islands to a relatively well performed island, so as to benefit reproduction from configurations of the well-performed islands.

2.3.2.4 Synchronous and Asynchronous Migration

It is a common practice to use multiple machines to speed up the island model. A parallel island model does not need to communicate as often as a traditional EA, which is important when the communication through a network is several orders of magnitude slower than within a single machine. The migration can be synchronous or asynchronous when the island model is implemented in parallel on multiple processors. Synchronous migration [36, 164–166] will not perform the computation such as crossover and mutation unless all islands finish exchanging

individuals. Therefore, all processors have to stall until the slowest one completes the migration, making it very inefficient and unscalable when a large number of islands are synchronised frequently by a centralised approach. Regarding the asynchronous migration, it performs the computation as soon as it finishes the exchange of individual, without taking into account the state of other islands.

Galeano and Fernández [167] compared the performance of island-based genetic programming (GP) with synchronous and asynchronous migration. Moreover, due to the close relationship between communication efficiency and number of processors, they also investigated how the number of islands impacts on the performance of parallel GP. In this work, the synchronous model will synchronise all islands when they share information; while the asynchronous model sends emigrants every a few generations and receives the immigrants whenever they arrive. Experiments indicated that if there are a small number of processors, synchronous model may perform better than asynchronous one because the latter model spend some time to check incoming message to receive in every generation while the former one only check after a few generations. However, when a large number of processors are utilised, asynchronous model performs significantly better than the synchronous model due to the high demands of synchronisation. Alba and Troya [168] also analysed the synchronous and asynchronous parallel GA. Experiments shows that asynchronous migrations effectively reduce the computing time compared to their equivalent synchronous versions for any interval and migration topologies. Regarding to the optimisation accuracy, both models can achieve similar numeric performance.

As the asynchronous island model offers better computational speed and similar solution quality with the synchronous one, it is becoming increasingly popular when implementing the island model in parallel. Liu and Wang [69] proposed a scalable parallel GA with an asynchronous migration which scales well on up to 16,384 CPU cores. To avoid the synchronisation among processors, an import pool is designed to store all incoming immigrants once they arrive. Similarly, a buffer-based asynchronous migration was also proposed by Märtens and Izzo [169] who designed an asynchronous island-based model by employing a list to store all emigrants. Kurose et al. [170] proposed an asynchronous migration EA by deploying the EA operations, fitness evaluation and communication on different processors. Consequently,

the efficiency of processors that work on computational tasks is hardly influenced because the synchronisation only exists among the processors that undertake the communication tasks. Izzo and Ampatzis [171] proposed an asynchronous island model by processing migration on a master processor and evolution tasks on clients. Unlike Zhan's work [36] in which master processor will not process migration unless all islands are ready, the master processor in this work performs asynchronous migration once any client is ready for importing or exporting individuals. Since the algorithm is implemented based on multi-threads programming library (POSIX THREADS) on a single machine, a mutual exclusion of access to the shared memory is designed to avoid the conflicts of the memory access.

2.3.3 GPU-based Parallel EAs

In recent years, GPUs have become a powerful and affordable computing device that can support general-purpose massive data-parallel computation. Nowadays, modern GPUs have empowered numerous personal computers (PCs), making it accessible for many researchers. As a dominant of parallel computing platforms and programming models, CUDA enables dramatic increases in computing performance by harnessing the power of the NVIDIA's GPU. As a result, many EAs that were sequentially implemented on central processing units (CPUs) have been re-designed and implemented based on CUDA, achieving remarkable speedups.

2.3.3.1 Parallel EAs on a Single GPU

DE was first implemented on GPU by Veronese and Krohling [74], Zhu [73], and Zhu and Li [172]. The GPU-based implementations are test on several numerical test problems and achieved up to 34 speedup comparing with sequential DE on CPU. Krömer et al. [173] implemented DE on GPU targeting at fully occupying the device. The experiments were tested on test problems and achieved up to 9.7 times speedup comparing with sequential DE. Qin et al. [174] proposed an improved GPU-based implementation of DE. This work considered the time of kernel launching and thus merged several kernels into one single kernel. Moreover, the configurations of program was automated decide by the device and several streams are used to increase the overlap between different kernels. Experiments showed the improved GPU-based

DE had a remarkable improvement compared to the original GPU-based DE based. Wong et al. [79] introduced the cuSaDE that implements the SaDE on GPU with CUDA. In this work, GPU conducted the time-consuming learning operator and the DE operators as well. Some benchmark functions were tested with different population sizes and problem dimensions. Results indicated that cuSaDE achieves a better speedup with a larger population size and problem dimension. Fabris and Krohling [175] proposed a co-evolutionary variant of the DE for max-min optimisation problems. In this work, DE maintained 2 populations for synchronisation and achieved a up to 6.33 speedup meanwhile obtained a promising solution. A more general framework was designed for reducing the difficulties for implementation on GPU by Arabas [176] who developed a more general framework of DE implemented on GPUs. In this work, a universal platform Easy Specification of Evolutionary Algorithms (EASEA) for GPU-based DE was introduced, in which just the fitness evaluation was conducted on GPU to reduce the difficulties of implementation on GPU. The implementation of fitness evaluation was the only task conducted by users which makes developing a GPU-based DE easier.

PSO was implemented and executed in parallel on GPU shortly after the inception of the platform. Rabinovich et al. [75] implemented a Gaming PSO on GPU in a single kernel. In this kernel, each thread was launched with each element of particle. The authors obtained a remarkable speedup on an NVIDIA GTX456 GPU. Roberge and Tarbouchi [76] proposed GPU-based PSO with curand library for generation of pseudorandom numbers. Several kernels are launched, in which one block handles one particle and one thread works for one element of one particle. Rosenbrock function was used as the test function and the speedup is 215 times faster than the CPU-based PSO. GPU-based PSO is also used in work of Roberge [177] for 3D pose estimation and a 140 times speedup over a sequential implementation is achieved. Reguera-Salgado and Martin-Herrero [178] applied thrust library to generate ortho images and reduced the execution time of GPU in less than 4 minutes. Platoš et al. [179] introduced a GPU-based PSO for document classification. Some widely used data mining benchmarks are employed and 2.5 to 10 speedups are achieved. Some promising variants of PSO are also employed and implemented on GPU. Zhang and Seah [180] employed a GPU-based niching PSO with local search. Kernels launched by the proposed algorithm contain 64 threads in one block

and up to 30 speedups are achieved. Nobile et al.[181] implemented a multiple-swarm PSO on GPU, which launched a kernel with one thread handling one particle in PSO operators and achieved 24 speedup compared to the CPU-based implementation. Sharma et al. [182] implemented a modified PSO based on GPU which achieved a 40 times speedup. Chen et al. [183] proposed a Latin Hypercube design for PSO (LaPSO) whose kernel are launched with one thread for each particle, the achieved speedup reaches up to 51 times faster than the CPU-based implementation. Different with most work which CPU launch GPU kernel function in every generation, the work done by Wachowiak and Foster [184] introduced an implementation of which the GPU threads run for a certain iterations before synchronisation. The algorithm was used to solve several realistic problems with different characteristics including toy protein folding, logistic function (regression benchmark problem) optimisation.

Other EAs were implemented on the GPU in last years. Zhou et al. [78] solved the travelling salesman problem (TSP) using a tour construction and pheromone update stages with Ant Colony optimisation (ACO) on the GPU and achieved up to 8 times speedup. Tsutsui et al. [185] solved quadratic assignment problems (QAP) problems by GPU-based GA. Experiments claimed a speedup ration from 3 to 12 times faster than the sequential GA. Dawson and Stewart [186] solved the edge detection problems with GPU-based ACO and a promising data-parallel approach was introduced that maps individual ants to A thread warp. The ants-to-warp showed its advantage and achieved up to 7.8 times speedup when comparing with ants-to-thread and ants-to-block. Chitty and Darren [187] studied the method to maximally utilised the performance of device, this work exploited the fast on-chip memory(L1 cache and shared memory) of GPU. Experiments showed that a maximum performance is up to 36 billion genetic programming (GP) operations per second.

2.3.3.2 Parallel EAs on Multiple GPUs

Goli and Brown [188] described a heuristic searching algorithm based on Monte Carlo Tree Search on heterogeneous CPU/GPU platform. They demonstrated that their algorithm achieved remarkable speedups on 2 GPUs. Pablo Vidal et al. [83, 189] designed a novel implementation of a cellular GA (cGA) model for a multi-GPU platform. It performed with two NVIDIA GTX-

285 cards and achieved remarkable speedups with a large population (up to 262,144). Tsutsui and Fujimoto [84] implemented ACO with two different parallel models (the island model and master/slave model) on a PC which has 4 GTX 480 GPUs. The master/slave model showed promising speedup for large instances of QAP problems. Ha and Moon [87] tackled the problem of knowledge discovery in big financial time series with GP on up to 8 GPUs. Jaros [86] proposed a novel implementation of the island-based GA exploiting a multi-GPU cluster. The proposed algorithm was executed on up to 14 NVIDIA GTX 580 cards and achieved the overall performance of 5.67 TFLOPS. Ježowicz [85] described five different evolutionary-based approaches that solved the classification problem on the Anselm cluster with up to 16 NVIDIA Kepler K20. The proposed algorithm only utilised the GPU to calculate the cost functions for the particles.

2.4 Modern High Performance Computing (HPC)

2.4.1 Overview

Currently, the further more computing power is always highly desired since the scale of real-world problems keep increasing. Therefore, as parallel computing platforms that can scale to a large number of processors, HPC [190–194] is the use of parallel processing techniques for solving complex computational problems and thus has become increasingly important in research, manufacturing and finance in recent years. Furthermore, dynamically provisioned and pay-as-you-go computing resources of HPC are now also offered by some cloud computing providers such as Amazon Web Services (AWS), Google Cloud Platform and Microsoft Azure. HPC cloud helps to reduce costs by providing CPU, GPU, and FPGA servers on-demand, optimised for specific applications, and without the need for large capital investments. HPC allows scientists or engineers to remarkably accelerate solving of compute-intensive problems by assembling a large number of processors, high-performance network, fast storage and large amounts of memory which can be seen at Figure 2.2.

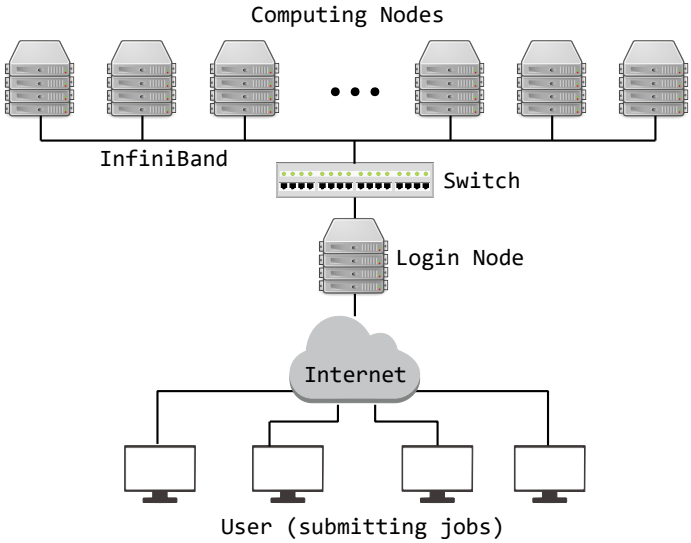


Figure 2.2: The architecture of HPC.

2.4.2 CPU-only HPC

A CPU-only HPC is a set of loosely or tightly connected CPU nodes that work together so that, in many respects, they can be viewed as a single system. Computer clustering relies on a centralised management approach which makes the nodes available as orchestrated shared servers. It is distinct from other approaches such as peer to peer or grid computing which also use many nodes, but with a far more distributed nature.

As shown at Figure 2.3 that presents the infrastructure of CPU-only HPC, the components of CPU-only HPC is the CPU compute node which has several CPUs, each of which usually possesses many computing cores, QPI that connects Intel CPUs at the same node, the fast local area networks (e.g. infiniBand) between different CPU compute nodes. Users submit jobs through the internet and login serve, while the jobs actually run at the CPU compute nodes. When utilizing CPU-only HPC, minimal changes are required to the existing source code of CPU programs, with the exception of possible modifications necessary for message passing. However, it is usually expensive to acquire sufficient computing power because numerous CPU cores are usually necessary to achieve significant computing power.

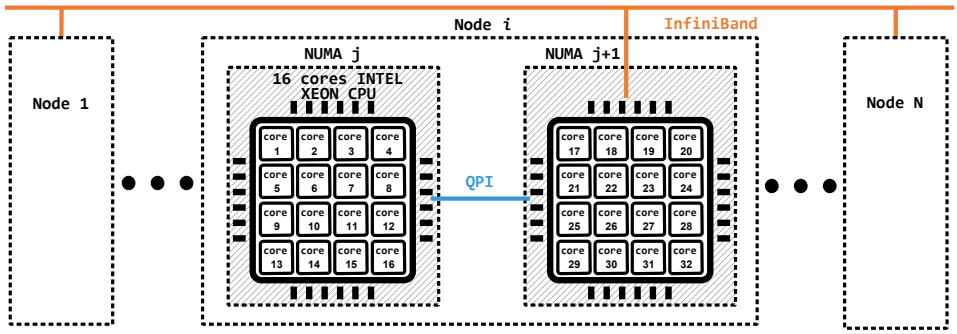


Figure 2.3: CPU-only HPC infrastructure.

2.4.3 GPU Computing

Modern Graphics Processing Units (GPUs), which are originally designed to support for computer graphics and gaming applications, can offer a considerably powerful platform in favor of massively parallel applications. Moreover, numerous PCs are equipped with affordable GPUs leading to powerful parallel platform accessible for resolving time-consuming applications. Currently, GPUs greatly outperform CPUs in both arithmetic throughput and memory bandwidth.

GPGPU has been a very active research topic in the last years, especially since computing frameworks such as CUDA or OpenCL were introduced. These platforms have allowed using the great computing capabilities of modern GPU for general purpose problems by using extensions of high level programming languages. OpenCL (Open Computing Language) [195–199] is the open, royalty-free standard for cross-platform, parallel programming of diverse processors found in personal computers, servers, mobile devices and embedded platforms. OpenCL greatly improves the speed and responsiveness of a wide spectrum of applications in numerous market categories including gaming and entertainment titles, scientific and medical software, professional creative tools, vision processing, and neural network training and inferencing. CUDA (Compute Unified Distributed Architecture) [200–209], which dedicated supports for modern NVIDIA's GPUs, is a parallel computing platform to improve the efficiency of general-purpose GPU computing. CUDA-C is an extension of the C programming language and requires less effort for programming on GPU when programmers are familiar with C/C++ language. This

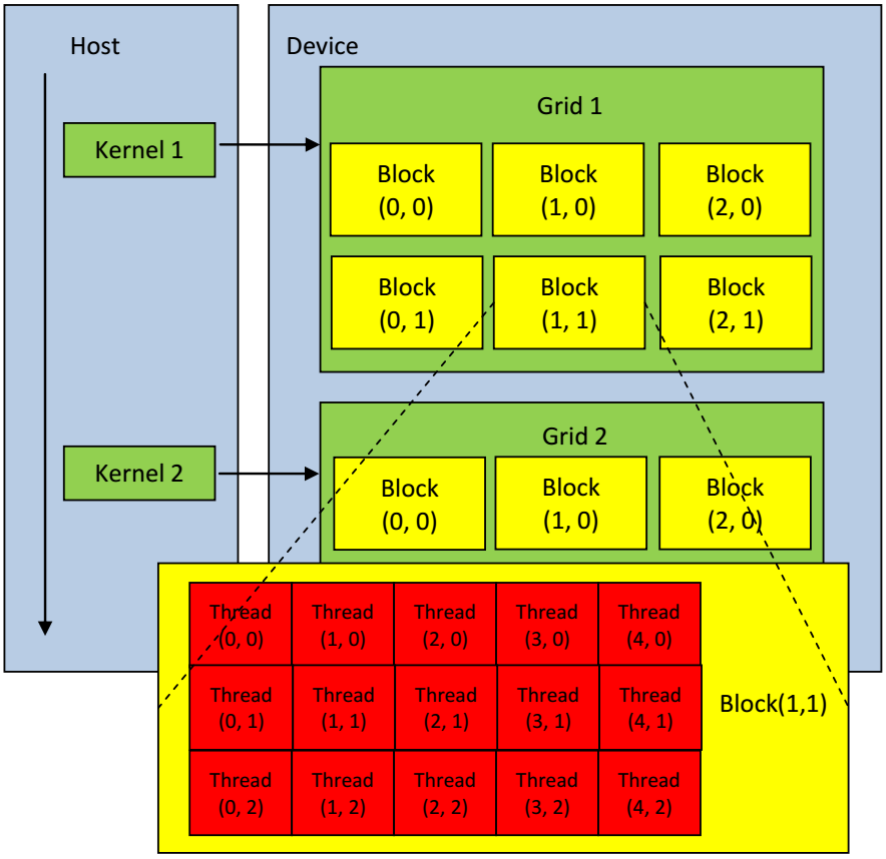


Figure 2.4: Threads batching in CUDA.

platform coordinates CPU and GPU, so-called host and device, into a heterogeneous computing system to make the best use of both of them. Although OpenCL promises a portable language for GPU programming on both AMD and NVIDIA cards, while CUDA only works on NVIDIA cards, OpenCL's generality may entail a performance penalty.

CUDA, which was first introduced by NVIDIA in November 2006, is a general purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU. CUDA comes with a software environment that allows developers to use C as a high-level programming language.

CUDA groups hundreds of SPs into several stream multiprocessors (SMs), each of which

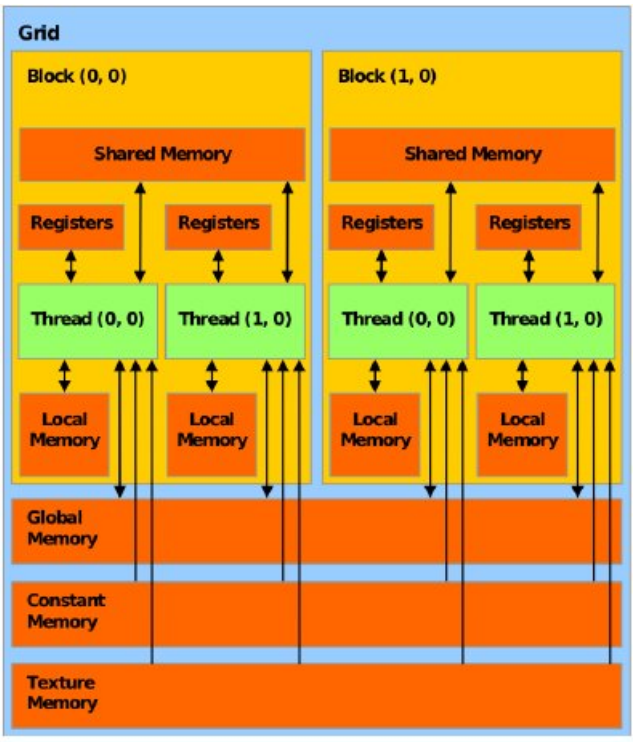


Figure 2.5: CUDA device memory model.

consists several SPs that share the on-chip control logic unit, shared memory with low latency and registers. SMs communicate with each other by global memory with high latency. From the birth of CUDA, 7 generations of hardware architectures has been released (Tesla, Fermi, Kepler, Maxwell, Pascal, Volta and Turing). The new generation always bring some breakthrough, i.e., the successful Kepler-class names the SMs as SMX and features a larger number of more powerful SPs; Kepler-class support dynamic parallel which also brings a better performance comparing with its previous generation (Fermi-class), Volta introduced the tensor core to improve the computing power for deep learning.

CUDA-C [200] is a C-based programming model for NVIDIA GPUs. This model unifies CPU and GPU, so called host and device, into a heterogeneous computing system to make the best use of both of them. The host code can contain arbitrary CC++ operations, data types, and functions while the device code can contain only a subset of operations and functions implemented on the device. The structure of CUDA program and the mapping of threads is

illustrated in Figure 2.4 from [210]. Some basic knowledge of CUDA-based programming is as follows:

- *Execution model:* CUDA-C consists of three types of functions: (1) host function, invoked and executed by CPU as same as C language. (2) kernel functions, invoked by CPU but executed on GPU. Kernel configurations such as dimensions of the block and the grid, the shared memory allocation and streams are associated and must specified within '<<<...>>>'. (3) device functions, invoked and executed by GPU. The kernel functions are device functions are tagged with 'global' and 'device' keywords respectively when declaring and defining functions.

- *Thread, block and grid:* hundreds or thousands of streaming processors (SP) are grouped into several streaming multiprocessors (SMs) on GPU) (presented at Figure 2.4). A thread is processed by each SP at one time and a large number of threads are executed by the same instruction at the same time in the single instruction multiple data (SIMD) mode, leading to the great parallelisation. These threads are grouped into several blocks, each one of which can be processed by a SM. A grid groups several blocks and is processed by a GPU. A block and grid are organised as one-dimensional, two-dimensional, or three-dimensional. The configurations of thread, block and grid are used in ¡¡¡...¿¿¿ execution configuration syntax mentioned above.

- *Memory types:*According to Figure 2.5 from [203] which presents the CUDA memory model, six types of memory with unique characteristics are provide by CUDA to better utilise the GPUs. (1) *registers* are independently accessed by each thread. They are the fastest on-chip memory but their number are very limited. (2) *shared memory* are jointly accessed by all threads in one block. They are slower than registers but also on-chip memory with low latency. They store more data than registers but still very limited. (3) *Local memory* accesses only occur for some automatic variables which are allocated with undetermined quantities or the required quantities are more than registers. (4) *constant memory* are read-only memory spaces accessible by all threads and is cached. (5)*texture memory* are read-only spaces accessible by all threads and is optimised cached for 2D

spatial locality. (6) *global memory* resides in device memory with the largest space but the slowest access speed.

2.4.4 GPU-enabled HPC

The HPC with millions of CPUs offers advantages in availability and extensibility, modularity and compatibility. However, these metrics do not tell the whole story. Specifically, since CPU is usually designed for general computing tasks instead of compute-intense tasks, numerous CPU cores are necessarily demanded to solve large-scale problems; as a result, it leads to the high expense.

Inspired by the incredible computing capability, many HPCs evolve from traditional clusters of homogeneous nodes (CPU-only) to clusters of heterogeneous nodes (CPU + GPU) [211, 211–213]. It can be observed from Table 1.1 that a half of top 10 HPCs are GPU-enabled. Especially, the latest and most powerful HPC (Summit at the Department of Energy's (DOE) Oak Ridge National Laboratory (ORNL)) is assembled with 26,136 NVIDIA Tesla V100 GPUs and provides with a performance of 122.3 petaflops on High Performance Linpack (HPL) benchmark. An example of infrastructure of GPU-enabled HPC is presented at Figure 2.6 based on the architecture of the GPU node at Australian National Computational Infrastructure (NCI)[1]. Three principal components are used in a GPU cluster: GPUs, interconnect and host CPUs. Since GPUs are designed to carry out a substantial portion of the calculations, high-end GPUs, such as the NVIDIA Tesla, are usually assembled by GPU-enabled HPC as the accelerators. In order to maintain a well balanced system, powerful PCIe bus and network interconnect (e.g. InfiniBand) with a low latency and high bandwidth are necessary. Furthermore, a many-to-one ratio of CPU cores to GPUs may be desirable to better utilise the GPU device in case some applications require extra CPU cores for specific operations.

Apart from the powerful computing capability, GPU-enabled HPC is better cost efficient (price/performance ratio) than CPU-only HPC. For example, when compared to AWS EC2 C4 instance (Intel Xeon E5-2666 v3), AWS EC2 P2 GPU instance (NVIDIA Tesla K80) only charges around 1/5 to achieve the similar performance on the Linpack benchmark [214]. In-

[1]https://opus.nci.org.au/display/Help/GPU+User+Guide

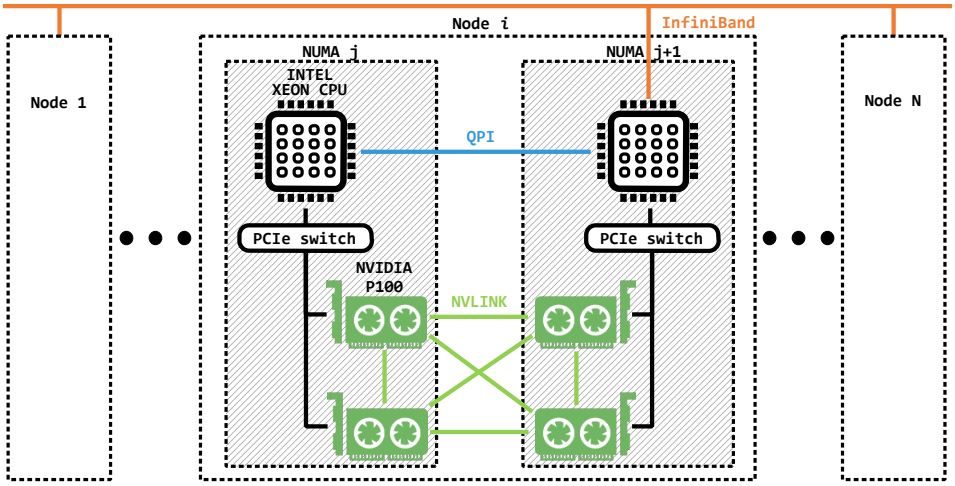


Figure 2.6: Infrastructure of GPU-enabled HPC (NCI).

spired by the attractive flops/dollar ratio and the incredible growth in the speed of modern GPUs, GPU-enabled HPC have become an ideal platform for scientific computing. For instance, the work [215] trained a deep neural network with 1 billion parameters by 3 GPU nodes in a couple of days and scale to networks with over 11 billion parameters with 16 GPU nodes.

Chapter 3

Study on the Effect of Large Population Size in EAs

3.1 Introduction

The increasingly complex and large-scale problems bring the rapidly rising solution spaces. Under this circumstances, many researchers [46–53] have started to study and to employ a large population to improve the searching capabilities of EAs for complex problems. So far, many theoretical works [54–59] have proven the benefits of a large population and some empirical studies [52, 53, 67, 131–145] are also conducted as a necessary supplement to these theoretical findings. However, a few researchers [60, 149–159] still insist that applying a large population does not bring any benefits but require more computational budget. Therefore, it is necessary to comprehensively examine the benefits of a large population by experiments taking into account that existing empirical works are limited in one or more of the following aspects:

- Focused on limited algorithms. Many works only examined the benefits of a large population based on limited series of algorithms (e.g. [135, 136, 141–143] based on GA and [137, 140] based on DE). However, comprehensively investigating the performance of a large population requires taking into account EAs with different searching patterns.
- Experimented with relatively small population sizes. Many works [52, 53, 131, 134–

137, 140] increased the population from a small size (e.g. 20 or 50) to less than or around 1000. However, several hundreds of individuals may not be adequate for increasingly complex real-world problems.

- Tested on specific or simple optimisation problems. Some works [138, 139, 143] only focused on the test problems in their specific domains and failed to promote universal conclusions. Some others [52, 53, 131, 132, 132–137, 140, 143] used a few famous benchmark functions for better applicability. However, these works are very limited in terms of both the number and difficulties of test problems. As a result, they cannot fully examine the potentiality of a large population.

- Performed the experiments with insufficient fitness evaluations (FEs) on sequential machines. So far, most works [53, 131–144] only implemented the algorithms in sequential. Due to the insufficient computing power, they are only able to offer a limited FEs, that are far from adequate to converge a large population.

Aiming to address these four issues, we systemically investigate the impacts of a large population in this chapter. Specifically, we select two state-of-the-art EAs that rank top at CEC competitions and three classic EAs (GA, PSO and DE) that have significantly different searching patterns. We also use eight difficult problems (composition functions f_{23}-f_{30} of CEC2014 benchmarks) that are never successfully solved and use three dimensions (10, 30 and 50) for each problem, which compose totally 24 problem instances. Furthermore, we implement three classic EAs in parallel on the GPU to achieve considerable FEs in a reasonable time. Experiments using population sizes from 64 to 4096 demonstrate that a large population can help EAs find better solutions on difficult problems if adequate FEs are provided. Moreover, we also present that a larger population can offer better parallelism and achieve better speedups when implemented on parallel computing platforms.

The rest of the chapter is organised as follows. In Section 3.2, we provide an overview of the experimental methodology. In Section 3.3, we present the experimental results and illustrate the benefits of a large population. Section 3.4 concludes this chapter.

3.2 Methodology

In this section we provide a detailed description of the experimental design. We first study on selection of difficult problems in order to examine the potentiality of large population on complex problems. Then we study on selection of several representative EAs in order to generalise the benefits of large populations.

3.2.1 Selection of Highly Complex Optimisation Problems

In order to demonstrate the potentiality of EAs on solving complex problems, we study on selecting some difficult problems from famous benchmarks which are never successfully solved by state-of-the-art EAs. In recent years, many real-parameter optimisation problems are proposed as ideal benchmarks to test the novel optimisation algorithms, i.e., the CEC [88] and BBOB [216]. In order to comprehensively test the performance of algorithms, benchmarks usually include various kinds of problems, for example, CEC2014 [88] comprises unimodal functions, simple multimodal functions, hybrid functions and composition functions.

In this work, difficult composition functions [217] are selected as the test problems to examine the performance of selected algorithms. The selected tested functions assembles several basic famous benchmark functions such as Rosenbrock, Griewank and Rastrigin to construct a series of very difficult composition functions. These composition functions are very deceptive compared to basic functions mentioned above because they have only one global optima but many local optima. Furthermore, these composition functions even employ gaussian functions to blur the landscape of each assembled function. The composition functions are asymmetrical multi-modal problems with different properties on different areas. They are minimisation problems defined as following:

$$min(F(x)), x = [x_1, x_2, \ldots, x_D]^T$$

where the construction of $F(x)$ are presented in [217]

$$F(x) = \sum_{i=1}^{N} \{\omega_i \cdot [\lambda_i g_i(x) + bias_i]\} + F^*$$

$g_i(x)$ can be a basic function, N is the number of basic functions, $bias_i$ defines the bias for each basic function: λ_i used to control each $g_i(x)$'s height and ω_i is the weight value for each

$g_i(x)$, it is calculated as $\omega_i = w_i / \sum_{i=1}^{N} w_i$, where w_i is defined as below:

$$w_i = \frac{1}{\sqrt{\sum_{j=1}^{D}(x_j - o_{ij})}} exp(-\frac{\sum_{j=1}^{D}(x_j - o_{ij})^2}{2D\sigma_i^2})$$

o_{ij} is new shifted optimum position for each $g_i(x)$ which defines the global and local optima's position, σ_i is used to control each $g_i(x)$'s coverage range, a small σ_i gives a narrow range.

In this chapter, we select 8 composition test functions ($f_{23} - f_{30}$) from CEC2014 [88] benchmarks as our test problems. The function value of the global best of each test function is 0 and thus these test functions are all minimizing problems. Each function is tested with 3 dimensions which are $D = 10, 30$, and 50, and each dimension is bounded within [-100, 100].

3.2.2 Selection of Representative EAs

In order to demonstrate that a large population is a common method to improve the performance of EAs, both the state-of-the-art and classic EAs are applied in this work. The classic EA applies some basic and simple mechanisms inspired by biological evolution, such as reproduction, mutation, recombination, and selection. In this work, we investigate the impacts of large population on three famous classic algorithms (DE, GA and PSO). Regarding the state-of-the-art algorithms, we survey the top-ranked state-of-the-art algorithms [218, 219] of famous CEC real-parameter single objective competitions from 2014 to 2017. There is an interesting phenomenon that most algorithms utilised dynamic population sizing methods [220–223] to effectively combines the advantages of both a large population's exploratory power and a small population's convergence speed. Here, we select two DE-based state-of-the-art algorithms L-SHADE [218] and jSO [219] which ranked 1^{th} in CEC2014 and 2^{nd} in CEC2017 competition respectively. Both algorithms apply a dynamic population sizing method LPSR (linear population size reduction) which compromises extra computing efforts and exploration ability by linearly reducing the population size during the increasing of generations. In this work, we compare the original state-of-the-art EAs with dynamic population sizing methods to their counterparts with a consistent large population to demonstrate that a consistent large population perform not worse or even better than dynamic population sizing methods regardless of the computing budget.

3.3 Experimental Results

In this section, we present the experimental analysis of the performance of EAs with a large population on solving complex problems in terms of effectiveness and time-efficiency. The experiments are started with analyzing the effectiveness of a large population on state-of-the-art EAs; namely, we compare the solution quality achieved by original L-SHADE and jSO with the population reduction scheme to their counterparts with a consistent large population. Then DE, GA and PSO are tested to further present whether classic algorithms with a larger population can benefit from larger populations. After that, we examine the time-efficiency of EA with a large population based on a single GPU.

3.3.1 Experimental Settings

Table 3.1: Selected algorithms and configurations.

Name	Configurations
L-SHADE	configured as [218]
jSO	configured as [219]
DE	current-to-rand/1/bin [101], $CR = 0.9$, $F = 0.5$ [47]
GA	binary tournament selection [224], linear crossover [119] elitist model [225], $CR = 0.6$, $MR = 0.001$ [226]
PSO	C_1=C_2 = 1.49445, $W = 0.729$ [48]

To avoid losing the generality, default configurations except population size are applied for all test algorithms and listed in Table 3.1. Since parallel EAs can offer considerable FEs to converge a large population, the maximal FEs are set as $D \times 10^6$ which is a hundred times bigger than the suggested $D \times 10^4$ in CEC2014 [88]. Each problem is tested with 3 dimensions ($D = 10, 30$ and 50). Each algorithm run 15 independently with different random number seeds on each problem and dimension for statistical analysis.

All experiments are conducted on NCI HPC. The sequential implementations run on NCI CPU node and the parallel implementations run on a single GPU at NCI GPU node[1].

[1]https://opus.nci.org.au/display/Help/GPU+User+Guide

3.3.2 Effectiveness of EAs with a large population

In order to comprehensively present the effectiveness of EAs with a large population, the comparisons are divided into two main parts. Firstly, we show that these state-of-the-art algorithms achieve better effectiveness when LPSR is replaced by a consistent large population. Secondly, we present that classic EAs can also benefit from a larger population by comparing themselves with smaller population sizes.

The effectiveness in this work is evaluated by comparing solution quality of a large population to the one of a small population. The solution quality is defined as the function error values (FEVs) [88], which presents the difference between the global optimum and the function values of the best found solution. The definition of FEVs is given as

$$FEVs = f(x) - F^*$$

where $f(x)$ is the fitness value of the best solution x found by EAs, and F^* is the known global optimal of the solution space. The results are shown with the mean values of 15 independent runs and statistically analyaed based on the Wilcoxon rank-sum test [227] with the significant level of 0.05.

3.3.2.1 State-of-the-art algorithms

Table 3.2: Mean FEVs of L-SHADE with a consistent population and LPSR.

D	NP	f_{23}	f_{24}	f_{25}	f_{26}	f_{27}	f_{28}	f_{29}	f_{30}
10	180 to 4	**329.46**	110.35	**112.43**	100.08	74.57	**375.45**	222.20	462.92
	180	**329.46**	**107.06**	**131.56**	**100.03**	**74.30**	**380.10**	**222.03**	**462.34**
30	540 to 4	**315.24**	**224.04**	**202.58**	100.16	**300.00**	**826.04**	**716.70**	**1040.81**
	540	**315.24**	**224.98**	**202.58**	**100.08**	**300.00**	**826.12**	**716.20**	**1062.49**
50	900 to 4	**344.01**	**275.11**	**205.30**	100.25	**333.10**	**1087.36**	**789.87**	**8682.94**
	900	**344.01**	**274.93**	**205.30**	**100.11**	**323.90**	**1105.61**	**807.30**	**8655.35**

Firstly, we compare the effectiveness of large population size on state-of-the-art EAs (L-SHADE and jSO). The Mean FEVs achieved by L-SHADE and jSO are presented in Table 3.2 and Table 3.3. The bold values in the table are statistically better than the normal values.

Table 3.3: Mean FEVs of jSO with a consistent population and LPSR.

D	NP	f_{23}	f_{24}	f_{25}	f_{26}	f_{27}	f_{28}	f_{29}	f_{30}
10	182 to 4	**329.46**	112.50	123.44	100.11	48.23	**378.95**	221.79	465.06
	182	**329.46**	**102.89**	**106.59**	**100.04**	**42.23**	**369.78**	**221.21**	**464.12**
50	466 to 4	**315.24**	**205.78**	**202.55**	100.25	**300.00**	**824.95**	715.89	767.38
	466	**315.24**	**201.45**	**202.54**	**100.08**	**300.00**	**814.01**	**714.60**	**616.84**
30	692 to 4	**344.01**	**271.26**	**205.06**	100.37	308.88	**1108.04**	814.16	**8269.82**
	692	**344.01**	**271.56**	**204.97**	**100.11**	**302.23**	**1087.71**	**778.06**	**8181.14**

According to Table 3.2 and Table 3.3, L-SHADE with a consistent large population performs significantly better than original L-SHADE on 7 out of 24 test problems (eight functions for three dimensions) and they perform similarly on the rest 17 problems. Regarding jSO with a consistent large population size, it performs significantly better than original jSO on 12 problems and does not lose in any problem. Summarily, with a sufficiently FEs, state-of-the-art EAs with a consistent large population size can help achieve better solutions than LPSR.

3.3.2.2 Classic EAs

Table 3.4: Mean FEVs of DE with NP=64, 256, 1024 and 4096.

D	NP	f_{23}	f_{24}	f_{25}	f_{26}	f_{27}	f_{28}	f_{29}	f_{30}
10	64	329.55	106.30	154.84	**100.03**	143.94	**405.99**	244.40	**539.13**
	256	**329.46**	111.25	164.14	**100.03**	209.16	**439.62**	228.73	618.06
	1024	**329.46**	109.41	138.83	**100.02**	26.72	**432.24**	**218.96**	**513.07**
	4096	**329.46**	**103.57**	**125.79**	**100.03**	**0.07**	**415.59**	**206.80**	**523.35**
30	64	341.28	232.46	206.95	114.63	489.80	1000.7	3341.6	4227.7
	256	315.74	**223.58**	203.67	106.76	320.26	835.96	1026.6	659.23
	1024	315.25	**223.33**	202.64	**100.09**	**300.00**	**813.13**	688.43	**378.35**
	4096	**315.24**	**223.13**	**202.59**	**100.11**	**300.00**	**808.04**	**567.89**	**376.45**
50	64	492.05	285.71	220.44	187.45	899.80	1985.6	6.7×10^5	8.6×10^4
	256	357.76	270.26	210.72	180.23	358.80	1262.4	1765.5	1.0×10^4
	1024	344.63	268.18	205.59	**101.14**	**300.00**	1140.2	887.59	8714.0
	4096	**344.17**	**267.52**	**204.91**	**100.20**	**300.00**	**1095.4**	**824.15**	**8297.8**

We have shown that a large consistent population contributes to L-SHADE and jSO in

Table 3.5: Mean FEVs of GA with NP=64, 256, 1024 and 4096.

D	NP	f_{23}	f_{24}	f_{25}	f_{26}	f_{27}	f_{28}	f_{29}	f_{30}
10	64	329.47	134.81	186.62	100.30	293.35	545.24	**1.4**$\times 10^5$	1086.25
	256	329.47	125.33	187.89	100.20	209.34	506.63	**395.17**	1117.65
	1024	329.47	120.71	160.64	**100.09**	4.40	451.18	**368.87**	966.87
	4096	**329.46**	**113.49**	**130.99**	**100.09**	**3.08**	**387.98**	**374.90**	**801.87**
30	64	315.35	231.11	213.75	140.32	725.21	1521.2	1359.3	2892.5
	256	315.28	230.52	212.30	107.06	589.70	1372.7	1152.4	3032.7
	1024	**315.25**	229.50	209.81	100.37	428.65	1159.3	**1049.4**	2788.3
	4096	**315.25**	**227.05**	**204.41**	**100.25**	**403.09**	**1019.6**	**957.09**	**2168.1**
50	64	344.02	263.98	228.87	180.48	1176.0	3006.6	1802.9	**1.2**$\times 10^5$
	256	344.02	261.53	225.42	166.96	1151.6	2479.7	1825.1	**1.3**$\times 10^5$
	1024	**344.01**	**262.53**	**210.01**	113.76	1132.9	2744.3	**1377.8**	**1.3**$\times 10^5$
	4096	**344.01**	**263.07**	**200.01**	**100.39**	**1019.6**	**2158.0**	**1148.1**	**1.3**$\times 10^5$

Table 3.6: Mean FEVs of PSO with NP=64, 256, 1024 and 4096.

D	NP	f_{23}	f_{24}	f_{25}	f_{26}	f_{27}	f_{28}	f_{29}	f_{30}
10	64	**329.46**	122.09	191.95	100.12	262.64	477.53	**394.58**	899.96
	256	**329.46**	119.17	188.98	100.08	235.99	446.81	**290.21**	**837.49**
	1024	**307.49**	116.76	**147.94**	**100.06**	80.28	**436.67**	**304.04**	**805.50**
	4096	**307.49**	**110.65**	**135.72**	**100.04**	**0.24**	**412.14**	**287.08**	**630.86**
30	64	315.27	**236.59**	210.82	106.98	654.66	1578.2	**4.2**$\times 10^6$	3525.5
	256	315.25	**230.51**	208.49	100.29	440.65	1393.3	**1562.3**	**2674.9**
	1024	315.25	**228.72**	**206.34**	**100.26**	**431.28**	**968.28**	**1509.7**	**2556.7**
	4096	**315.24**	**229.54**	**205.82**	**100.23**	**401.51**	**959.77**	**1274.1**	**2118.0**
50	64	344.65	284.28	224.68	153.76	1320.8	3685.7	4.7$\times 10^7$	3.0$\times 10^5$
	256	344.04	**277.53**	**217.20**	140.38	1095.6	2924.9	**8.5**$\times 10^6$	1.7$\times 10^5$
	1024	344.02	**276.14**	**215.40**	**120.41**	1059.3	2386.3	**5.1**$\times 10^6$	1.5$\times 10^5$
	4096	**344.01**	**276.22**	**214.63**	**100.38**	**871.58**	**1879.4**	**2101.1**	**1.3**$\times 10^5$

terms of effectiveness. In this experiment, we demonstrate that a large population contributes to not only DE-based state-of-the-art algorithms (L-SHADE and jSO), but also various classic EAs. Here, we compare the mean FEVs obtained by classic EAs (DE, PSO and GA) with different population sizes that range from small (NP = 64) to large (NP = 4096) at Table 3.4, Table 3.5 and Table 3.6. Results show that a large population size can achieve far better solutions than a small population on some problems; i.e., mean FEVs achieved by DE with

Table 3.7: Summarised statistical tests(+/≈/-) indicate that $NP = 4096$ performed significantly better (+), similarly (≈) or worse than $NP = 64$, 256 and 1024, respectively.

Algorithms	$NP = 4096$ vs. $NP=$	$D = 10$	$D = 30$	$D = 50$
DE	64	5 / 3 / 0	8 / 0 / 0	8 / 0 / 0
	256	4 / 4 / 0	7 / 1 / 0	8 / 0 / 0
	1024	3 / 5 / 0	3 / 5 / 0	6 / 2 / 0
GA	64	7 / 1 / 0	7 / 1 / 0	8 / 0 / 0
	256	7 / 1 / 0	7 / 1 / 0	8 / 0 / 0
	1024	6 / 2 / 0	6 / 2 / 0	6 / 2 / 0
PSO	64	6 / 2 / 0	6 / 2 / 0	8 / 0 / 0
	256	5 / 3 / 0	5 / 3 / 0	5 / 3 / 0
	1024	2 / 6 / 0	2 / 6 / 0	4 / 4 / 0

$NP = 4096$ on f_{29} for $D = 50$ reaches 824.15 that is much smaller than 6.7×10^5 achieved by DE with $NP = 64$.

Table 3.7 shows the aggregate results of statistical analysis of three algorithms by comparing the mean FEVs achieved by $NP = 4096$ with $NP = 64$, 256 and 1024 on 8 test problems for D=10, 30 and 50. According to Table 3.7, major observations are as follows:

- Classic EAs with a large population size ($NP = 4096$) perform better than them with smaller population sizes ($NP = 64$, 256 and 1024) for all dimensions. Specifically, regarding 24 test problems (8 test function and 3 dimensions), DE, GA and PSO with smaller population sizes cannot even perform significantly better than $NP = 4096$ on any problem. Moreover, results indicate that the larger difference of population size is, the larger improvement is achieved, i.e., if utilizing PSO on problems for $D = 10$, $NP = 4096$ can only win at two out of eight compared to $NP = 1024$, but the wins reach six compared to $NP = 64$.

- Benefits of a large population to classicc EAs become more remarkable when problem dimensions increase from 10 to 50. It is due to the fact that with the dimension increases, the problem could be more complicated and a larger population can avoid EAs to fall into a local optima easily.

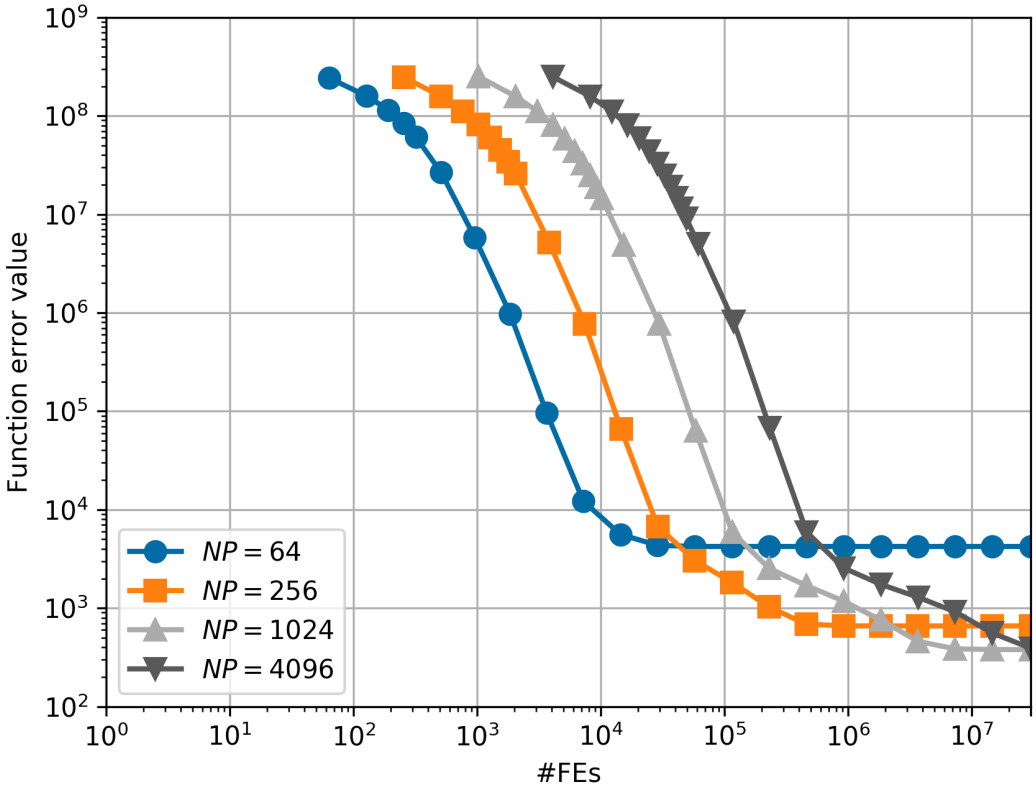


Figure 3.1: Mean convergence characteristics of DE with $NP = 64$, 256, 1024 and 4096 on f_{30} for $D = 30$.

In order to further study the reason why a larger population brings better solutions, we show the curve of convergence of DE with four different populations ($NP = 64$, 256, 1024 and 4096) on f_{30} for $D = 30$ as an example in Figure 3.1. It can be observed that:

- DE with a larger population converges slower than a smaller population, i.e., to reach the 10^5 function error values (FEVs), DE with $NP = 4096$ requires more than 10^5 FEs, while $NP = 64$, 256 and 1024 only need 2×10^3, $9 \times 10^3$and 2×10^4 respectively.

- DE with a larger population can eventually find better solutions if a large number of FEs are provided, i.e., DE with $NP = 64$, 256 and 1024 already converges at 4×10^3, 8×10^2 and 4×10^2 FEVs respectively, while DE with $NP = 4096$ has already achieved 4×10^2

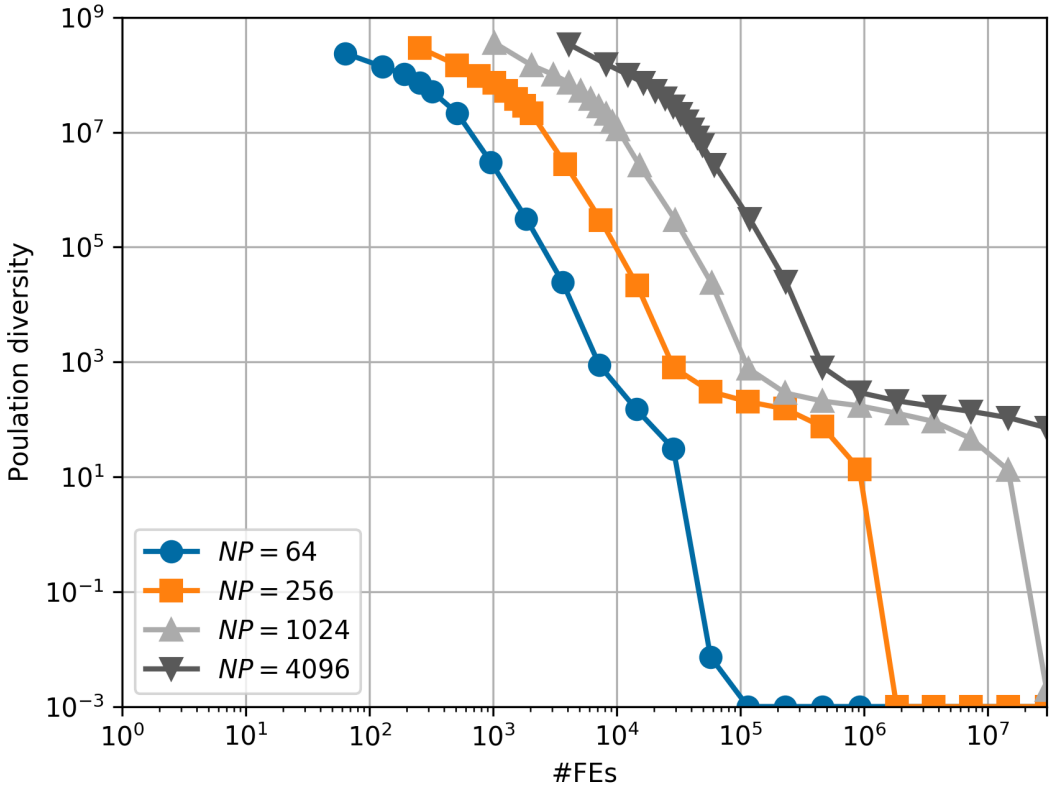


Figure 3.2: Mean population diversity of DE with $NP = 64$, 256, 1024 and 4096 on f_{30} for $D = 30$ (Population diversity smaller than 10^{-3} is recorded as 10^{-3}).

when it reaches the maximal FEs and its FEVs can be further improved if more FEs are provided based on the tend in this figure. Therefore, a large population can help find the better solutions but requires more computing budgets.

Figure 3.2 shows the population diversity that is defined as the standard deviation of fitness values of individuals in population [228]. It can be observed that the curve of diversity includes four stages as follows:

- Stage 1: in the beginning, DE with different population sizes has the similar behavior as their population diversities drop fast from 10^9 to 10^3. The reason of initially high diversity population is the randomly generated individuals that are uniformly distributed in the

entire searching space. Due to the employing of evolutionary operations, some diverse but bad individuals are replaced by better individuals that are more similar compared to the randomly generated ones.

- Stage 2: once population diversity reaches 10^3, it decreases gently for a while. It could be due to the process that DE searches the neighbor of many local optima but does not converge yet. According to Figure 3.2, a larger population size has a longer Stage 2, i.e., when $NP = 64$, DE stays in this stage from 7×10^3 to 2×10^4 FEs, while $NP = 1024$ stays in this stage from 10^5 to 1.5×10^7 FEs. $NP = 4096$ stops at this stage because the maximal fitness evaluation reaches.

- Stage 3: after the Stage 2, the population diversity rapidly decreases to 0. It is due to the reason that the population of DE converges to the best local optima.

- Stage 4: population diversity stays at 0 until the maximal FEs reach.

Summarily, DE with different population sizes mainly behaves differently in stage 2. A small population stays at Stage 2 quite short and its diversity decreases rapidly because the small population is hard to maintain the diversity in search space with many local optima. Consequently, it is easy to converge a small number of individuals to the best solution found so far. On the contrary, a large population stays at Stage 2 for a long period and diversity decreases gently because a large number of individuals can search the neighbor of many local optima concurrently and delay the convergence to some promising local optima.

3.3.3 Efficiency of EAs with a large population

We have illustrated the effectiveness of EAs with a large population regardless of the computing time. However, achieving a large number of FEs in a reasonable computing time is also significant if a large population is employed. This section examines the impacts of a large population on the computing efficiency. The computing efficiency can be measured by the speedup that is defined as

$$speedup = \frac{T_s}{T_{gpu}}$$

Table 3.8: Computation speed of sequential and parallel DE measured by time (hh:mm:ss) and the speedup of parallel DE with NP = 64, 256, 1024 and 4096 on f_{24} and f_{26} for D = 10, 30 and 50.

D	NP	elapsed time (hh:mm:ss)					
		sequential DE		parallel DE		speedup	
		f_{24}	f_{26}	f_{24}	f_{26}	f_{24}	f_{26}
10	64	0:00:55	0:04:35	0:00:15	0:00:32	3.6	8.4
	256	0:00:55	0:04:35	0:00:05	0:00:13	10.8	20.7
	1024	0:00:56	0:04:35	0:00:02	0:00:07	22.1	38.5
	4096	0:00:56	0:04:36	0:00:01	0:00:05	29.2	47.4
30	64	0:06:11	0:35:42	0:00:49	0:01:44	7.4	20.6
	256	0:06:11	0:35:44	0:00:16	0:00:40	22.2	53.3
	1024	0:06:11	0:35:45	0:00:08	0:00:22	45.1	96.6
	4096	0:06:12	0:35:46	0:00:06	0:00:18	60.3	115.0
50	64	0:20:03	1:55:50	0:01:13	0:03:45	12.4	30.7
	256	0:20:04	1:55:51	0:00:30	0:01:36	34.2	71.9
	1024	0:20:06	1:55:55	0:00:16	0:00:58	62.2	119.7
	4096	0:20:08	1:55:57	0:00:11	0:00:50	77.2	137.8

where T_s and T_{gpu} are the runtime of sequential and GPU-based parallel implementation respectively.

In this experiment, we select two functions as the test problems which are the fastest (f_{24}) and the slowest (f_{26}) in eight composition functions ($f_{23} - f_{30}$). Table 3.8 shows the runtime of sequential DE on a single CPU core; as well as the runtime and speedups of parallel DE on a single GPU. The time presented in this table is calculated based on a single instance of problem. The major observations are as follows:

- When dimensions increase from 10 to 50, the runtime of sequential DE increases rapidly, i.e., sequential DE requires only 5 minutes on f_{26} for $D = 10$ but around 2 hours for $D = 50$. Therefore, when solving complex and complicated problems with a high dimension, sequential EAs with a large population become impractical due to the requirement of an unreasonable computational budget.

- Parallel DE runs significantly faster than sequential DE, especially for the more time-

consuming problem. For example, parallel DE achieves 77.2 speedups on f_{24} for $D = 50$ but achieves 137.8 speedups on f_{26} .

- Given a problem, a larger population always brings larger acceleration. For example, regarding to f_{26} on $D = 50$, the speedup is 30.7 for $NP = 64$ and 71.9 for $NP = 256$.

Here, the main finding is that increasing population can always improve the speedups. Given a specific problem dimension that corresponds to a specific maximum number of FEs, the population size determines the total number generations (main loops) and accordingly the execution times of GPU kernels functions involved in the main loop. As the population size increases, the number of main loops decreases, which potentially reduces the total computing time. Therefore, a larger population can benefit computing efficiency if EAs are implemented in parallel.

3.4 Conclusions

This chapter experimentally studies the performance of EAs with a large population on solving complex and complicated problems in terms of solution quality and runtime. Specifically, we apply two state-of-the-art algorithms and three classic EAs on eight difficult composition problems to investigate the ability to search good solutions of EAs with a large population. Experiments showed that EAs with a large population can achieve significantly better solutions than those of EAs with a small population, as well as better speedups when implemented in parallel.

Chapter 4

SPEO based on CPU-only HPC ($\text{SPEO}_{\text{HPC}_{\text{cpu}}}$)

4.1 Introduction

Chapter 3 has shown that a large population can significantly improve the solution quality of EAs. However, it also mentioned that parallel computing platforms and parallel implementations of EAs are necessary to achieve sufficient FEs in a reasonable time. Therefore, this chapter studies designing a generic scalable framework of parallel EAs which has outstanding scalability and wide applicability. Specifically, the proposed framework is designed to accelerate various classic or state-of-the-art sequential EAs by utilising numerous CPU cores at CPU-only HPC.

The proposed framework is based on the island model [14, 23–25], which has been employed by many parallel EAs [35, 37, 63–70]. However, most of existing works are designed to efficiently use only a small number of islands to avoid the serious communication congestion which significantly reduces the computational efficiency. Although a few studies [35, 37, 67–70] are able to use a large number of islands, these approaches have to employ sparse communication topology. Have been proven by many works [66, 71, 72], these approaches reduce the communication workload at the sacrifice of searching capability but also bring the worse solution quality. Aiming to solve this issues, we propose the SPEO based on CPU-only HPC

($\text{SPEO}_{\text{HPC}_{\text{cpu}}}$) with a buffer-based asynchronous migration to avoid the serious communication congestion and to significantly improve the computing efficiency. Then, we implement the proposed framework based on a standard DE and examine its performance on eight composition functions of CEC2014 benchmark at the Australian National Computational Infrastructure (NCI) platform using up to 512 CPU cores. Experimental results demonstrate that $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ is very scalable even a dense topology is employed; namely, approximately linear speedups are achieved when a fully connected topology is used. The results also present that it not only increases the computational efficiency but also improves the solution quality when compared to a state-of-the-art island-based parallel EA.

The remainder of this chapter is organised as follows. Section 4.2 describes the design of the proposed $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$. Experimental results are presented in Section 4.3. Section 4.4 concludes this chapter.

4.2 The Proposed Method

In this section, we firstly introduce the framework of the proposed $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ and then we implement the framework based on a standard DE algorithm.

4.2.1 Framework

In order to efficiently utilise a larger number of CPUs at CPU-only HPC in a scalable way, the island model [14, 23] is chosen as the population-distribution model due to the following reasons:

- The island model can work with a larger global population by simply adding islands on more processors. It offers excellent scalability to solve increasingly complex problems.
- As different islands may have various searching behaviors, the island model can offer outstanding population diversity which is also the target of using a large population.
- Apart from the information exchange between islands, each island evolves itself independently as a common sequential EA. Thus, most existing sequential EAs can be simply

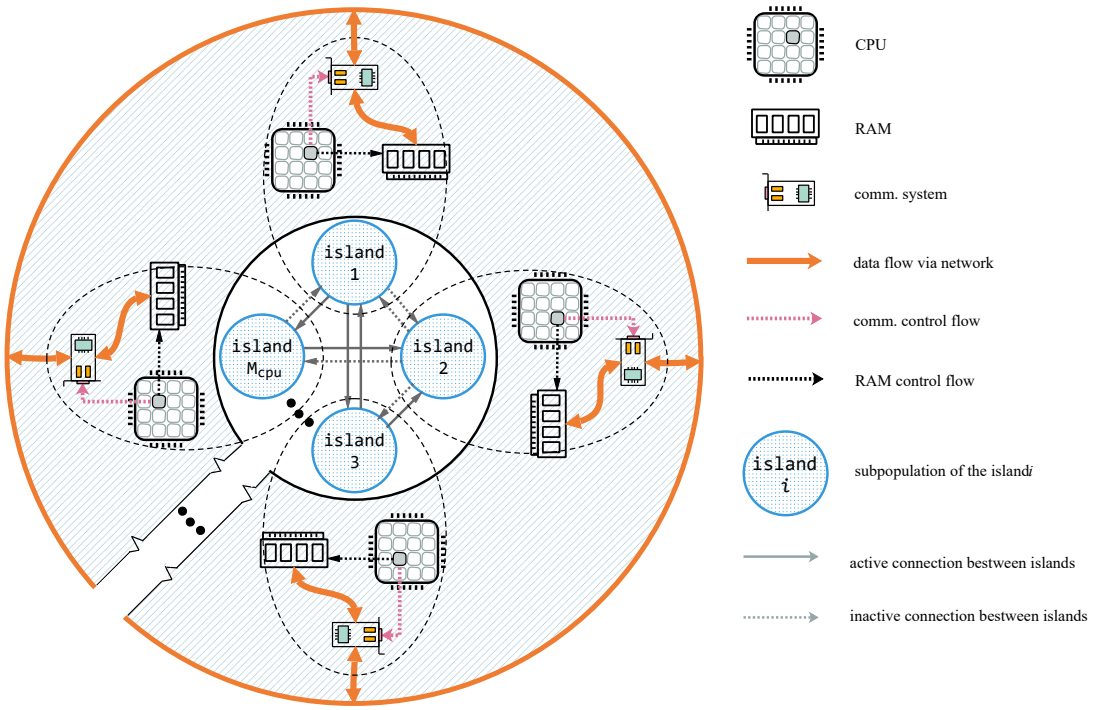


Figure 4.1: Deployment of $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ on CPU-only HPC.

accelerated by applying the proposed framework and deploying on many CPU cores.

- The performance of the island model is determined by the migration topology [66]. The island model can have various searching behaviors if different migration topologies are employed. For example, it has been proven that a dense migration topology is suitable to solve complex and difficult problems [66, 71, 72]. However, existing works fail to efficiently employ a dense topology over a large number of CPU cores because it brings a significant communication congestion.

4.2.1.1 $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ and the infrastructure of CPU-only HPC

The population distribution model of $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ is presented at the inside circle of Figure 4.1. Here, an island with N_s individuals are deployed to each of M_{cpu} CPU cores; thus the global population contains totally $NP = N_s \cdot M_{cpu}$ individuals. These islands exchange information via

a specific migration topology represented by solid lines in Figure 4.1. It also can be observed that all pairwise islands are connected by either the dotted and solid lines, which indicates that the proposed framework is not limited by a specific topology and supports even the most complex topology (fully connected).

The deployment of $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ on a common infrastructure of CPU-only HPC is presented at the outside ring with the shadow in Figure 4.1. Enclosed by the dotted ellipse, a CPU core, RAM and communication system undertake all operations to evolve a single island. Without loss of generality, the communication system can be a mixture of network (e.g. InfiniBand), inter-core communication, point-to-point processor interconnect (e.g. QPI) and some specific devices applied by CPU-only HPC.

The proposed framework is presented at Figure 4.2. Accordingly, the proposed framework is concurrently initialised on M_{cpu} CPU cores. Each CPU core performs EA operations to evolve its corresponding island and performs asynchronous migration to exchange information with other CPU cores. Each island is also enclosed with a buffer which temporarily stores immigrants sent from other islands. The island frequently updates itself by inserting some immigrants selected from the buffer. After all CPU cores terminate its tasks, a centralisation operation will be performed to summarise all islands and output the best solution found so far.

4.2.1.2 Buffer-based asynchronous migration

When more CPU cores are demanded, increasingly serious communication congestion will significantly reduce the computing efficiency; especially a dense migration topology is utilised. In this chapter, the proposed framework introduces a buffer-based asynchronous migration strategy to achieve the outstanding scalability. At Figure 4.2, the migration operation undertakes information exchange including three main aspects as follows:

- **Update island (green arrow)**: it updates the island when the current island seeks new information to better explore in the searching space. In this operation, several immigrants are selected from the buffer and are merged with the current island.
- **Import immigrants (yellow arrow)**: it receives newly arrived immigrants that are

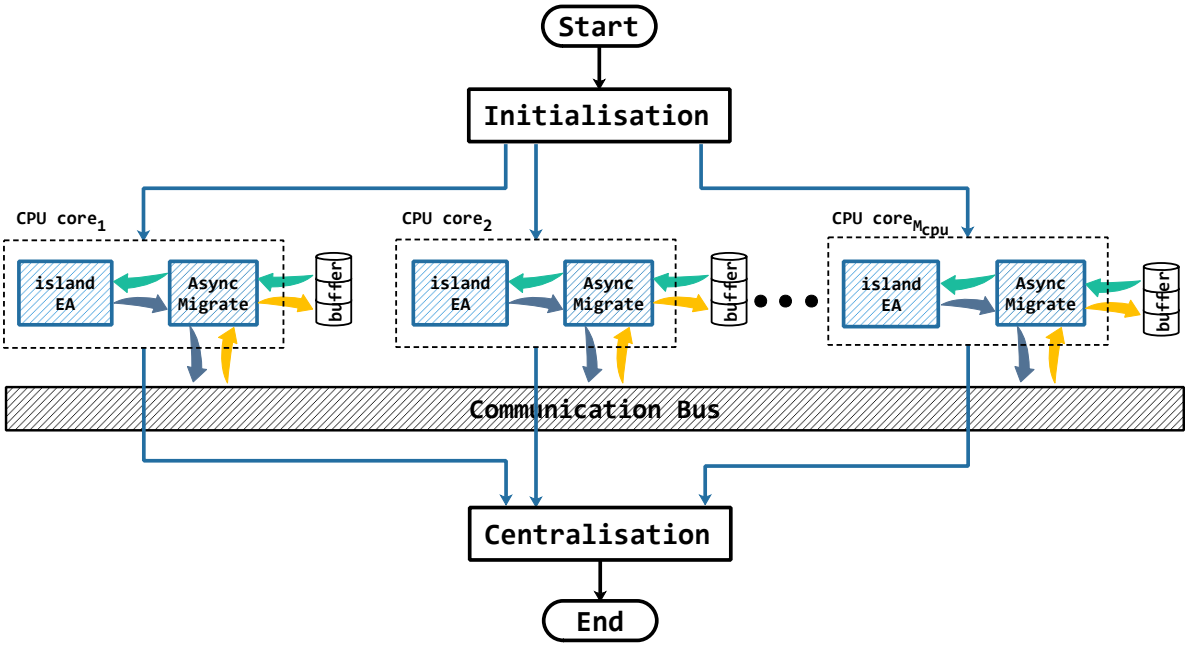


Figure 4.2: Framework of $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$.

sent from other islands. Instead of directly inserting into the island, immigrants are temporarily stored at the buffer based on a specific buffer management scheme.

- **Export emigrants (grey arrow)**: it selects emigrants from the current island and sends them to other islands via the communication bus. As an asynchronous migration, it will not stall for the success of sending; instead, it will immediately step into the further instructions.

As the key component of the $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$, buffer-based asynchronous migration improves the computing efficiency when a large number of CPU cores are utilised. Specifically, the migration is only performed based on the runtime status of the $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$; as a result, it will not wastes any computational budget on stalling for the communication congestion such as 1) waiting for availability of communication system to send emigration and 2) waiting for the incoming immigrations from other islands.

Algorithm 1: DE-based $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ on i^{th} CPU core, $i \in [1, M_{cpu}]$

input : {island size N_s, buffer capacity C_b, migration rate R_m, connection rate R_c, interval I}

Initialise island $\mathbf{P}_{i,0}$ with N_s individuals;
Initialise buffer $\mathbf{B}_i \leftarrow \emptyset$, $G = 0$;
while *the predefined termination criteria is not met* **do**
    `/* EA: Perform DE algorithm */`
    Perform DE/rand/1/bin at $\mathbf{P}_{i,G}$ to generate trial vectors $\mathbf{T}_{i,G}$;
    Evaluate $\mathbf{T}_{i,G}$;
    Generate $\mathbf{P}_{i,G+1}$ based on $\mathbf{P}_{i,G}$ and $\mathbf{T}_{i,G}$;
    $G \leftarrow G + 1$;
    `/* Async migrate: Update island */`
    if $mod(G, I) = 0$ **then**
        Select the first $R_m * N_s$ immigrants from the buffer $\mathbf{B}_i$ and insert into island $\mathbf{P}_{i,G}$;
        Delete used immigrants from the buffer;
    end
    `/* Async migrate: Import immigrants */`
    while *immigrants arrive (*$mpi\_Iprobe() = 1$*)* **do**
        Receive arrived immigrants **IM** by $mpi\_Recv()$;
        `/* Diversity preserving buffer */`
        foreach im $\in$ **IM do**
            if $length(\mathbf{B}_i) < C_b$ **then**
                insert **im** into $\mathbf{B}_i$;
            else
                b $\leftarrow$ nearest immigrant in the buffer $\mathbf{B}_i$ to **im**;
                if b *is worse than* **im then**
                    Replace **b** with **im**;
                end
            end
        end
    end
    `/* Async migrate: Export emigrants */`
    r $\leftarrow$ randomly select $R_c * (M_{cpu} - 1)$ recipient islands;
    EM $\leftarrow$ select the best $R_m * N_s$ emigrants from $\mathbf{P}_{i,G}$;
    while *Communication system is available (*$mpi\_Test() = 1$*) and* **r** *is not empty* **do**
        Send **EM** to the first island in **r** by $mpi\_Isend()$;
        Delete the first island from **r**;
    end
end

4.2.2 Implementation of $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$

The $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ framework can be implemented by developers based on their specific demands. Algorithm 1 presents the implementation of the proposed $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ based on a standard DE. In this work, several key features of the implementation are described as follows:

- **Employing EA**: DE [47] is selected and implemented since it is an efficient and effective global optimiser in the continuous search domain.

- **Selecting emigrant and importing immigrant**: selecting emigrant represents the way of selecting emigrants from the current island and importing immigrant represents the way of merging immigrants with the current island. Here, both operations apply an elitism-based method; specifically, the best $R_m * N_s$ ($R_m \in [0, 1]$ is the migration rate) individuals are selected from the current island as emigrants and are sent to the recipients. Regarding immigrant importing, the $R_m * N_s$ immigrants are firstly inserted into the current island and then the best N_s individuals are selected from $N_s + R_m * N_s$ merged individuals as the new island.

- **Diversity preserving buffer management**: In order to avoid importing too many similar or bad immigrants into the island, this work implements a diversity preserving buffer (line 14 to line 23 at Algorithm 1) which is designed to only store high-quality or unique immigrants sent from other islands. Compared to traditional buffers such as first-in-first-out (FIFO) or best preserving, the diversity preserving buffer stores immigrants considering their characters both in solution and objective space. As a result, the imported immigrants are diverse and promising which furthest prevent the premature converging.

- **Dense migration topology**: Compared to existing island-based parallel EAs, the proposed framework can efficiently employing dense migration topologies. Here, we implement a random-based migration topology that is called improved dynamic migration topology which is extended from the standard dynamic topology [163]. The standard dynamic topology randomly selects only one recipient for each migration, while our proposed

dynamic topology randomly selects distinct $R_c * (M-1)$ recipients, where $R_c \in (0, 1]$ is the connection rate that indicates how dense the topology is. Therefore, the topology is able to vary from the sparsest (the standard dynamic topology, $R_c < {}^1/_{M-1}$) to the densest (the fully connected topology, $R_c = 1$) based on the actual demands.

The main body of $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ is implemented with C/C++ and the communication operations ("import" and "export") are implemented using the MPI message-passing programming model [229]. The non-blocking point-to-point communication API MPI_Isend() and MPI_Iprobe() are used for sending or checking messages, respectively. The blocking receiving API MPI_Recv() is employed to receive messages once MPI_Iprobe() indicates any message arrives. MPI_Test() is utilised to check whether the previous message is sent successfully.

4.3 Experimental Results

In this section, the performance of the proposed $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ is examined at following aspects:

- Whether the $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ is scalable with increasing CPU cores and whether the scalability benefits from the asynchronous migration.
- Whether the diversity preserving buffer outperforms traditional buffers on the solution quality and how the buffer capacity influences the solution quality.
- How the migration topology density influences the solution quality and whether the dense topology improves the solution quality of $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ compared to some traditional topologies.
- Whether the proposed $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ outperforms the state-of-the-art island-based EA in terms of the solution quality and the computational speed.

4.3.1 Test Problems

To examine the performance of $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$, 8 difficult test functions (complex composition function $f_{23} - f_{30}$ from CEC2014 [88] benchmarks) which are briefly introduced at Chapter 3.2.1

Table 4.1: Configurations of $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$.

Parameters	notation	Settings
Global population size	NP	8192
CPU cores	M_{cpu}	4, 8, 16, 32, 64, 128, 256, 512
Interval	I	100
Connection rate	R_c	0.1%, 1%, 2%, 5%, 10%, 25%, 50%, 100%
Migration rate	R_m	5%
Buffer capacity	C_b	16
EA		DE/rand/1/bin, CR=0.9 and F=0.5

are selected as our test problems. Each function has 3 dimensions which are $D = 10, 30$, and 50. Each dimension is bounded within [-100, 100]. 8 test problems are implemented with C/C++.

4.3.2 Experimental Settings

4.3.2.1 Configurations

Table 4.1 presents the configurations of $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$. In order to avoid losing generality, DE is configured with standard settings that are DE/rand/1/bin with $CR = 0.9$ and $F = 0.5$ based on the work [47]. Taking into account the demand of great number of FEs for converging a large population in a reasonable time, the total fitness evaluations are set large as $D * 10^6$.

4.3.2.2 Computing Platform

All experiments are conducted on CPU nodes at NCI that is Australia's most highly-integrated high-performance research computing environment and supercomputer. In this work, up to 512 CPU cores are utilised to comprehensively investigate the scalability of the proposed $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$.

4.3.2.3 Evaluations

The solution quality is measured by the function error values (FEVs) [88], which is the index presenting the difference between the global optimum and the function values of the best found solution. The definition of FEVs is given as

$$FEVs = f(x) - F^*$$

where $f(x)$ is the fitness value of the best solution x found by EAs, and F^* is the known global optimal of the solution space. The results are shown with the mean values of 15 independent runs and statistically analysed based on the Wilcoxon rank-sum test [227] with the significant level of 0.05.

The computational efficiency is measured by the speedups of the parallel implementation of the algorithm executed on the CPU-only HPC against its sequential counterpart executed on the single CPU core. The definition of speedup is given as

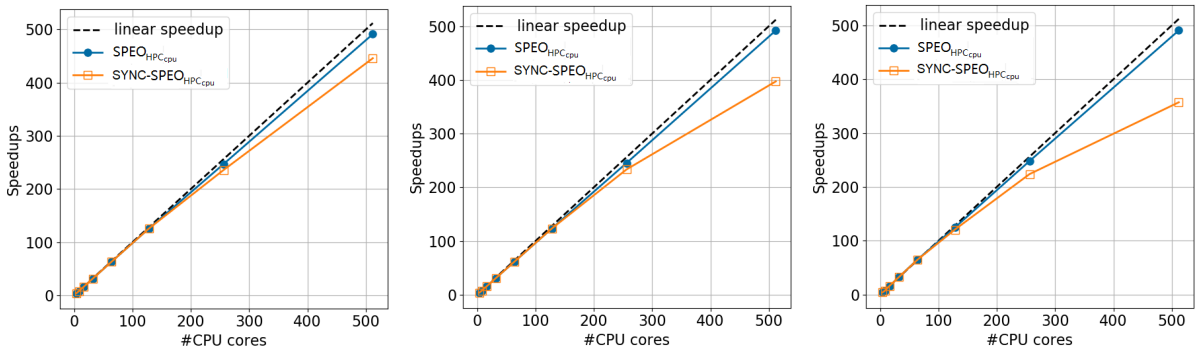
$$speedup = \frac{T_s}{T_{ncore}}$$

where T_s and T_{ncore} are the average execution time of the sequential and the parallel implementations over n CPU cores on all test problems, respectively.

4.3.3 Scalability Analysis

In Chapter 3, experiments illustrate that a significant computing budget is necessary to effectively converge a large population using extensive FEs, which brings long computing time if executed sequentially. In this section, we present that $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ with a large population ($NP = 8192$) can always achieve approximately linear speedups when increasing number of CPU cores are used.

Therefore, the strong scaling test [230, 231] in parallel computing is employed to evaluate the scalability of the proposed $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ framework. Extended by Liu's work [69], the strong scaling test can demonstrate how the execution time varies with the increasing number of devices when the global population size NP is fixed.



(a) $D = 10$ (b) $D = 30$ (c) $D = 50$

Figure 4.3: Average speedups of SPEO$_{\text{HPC}_{\text{cpu}}}$ with asynchronous or synchronous migration on 8 test problems for $D = 10, 30$ and 50 on up to 512 CPU cores.

The scalability of the SPEO$_{\text{HPC}_{\text{cpu}}}$ is demonstrated by comparing it to the linear speedups from 4 to 512 processors at Figure 4.3. It can be observed that SPEO$_{\text{HPC}_{\text{cpu}}}$ achieves approximately linear growth of speedups with the increasing number of CPU cores for all dimensions. Since more CPU cores results in more incoming immigrants and thus a larger cost on calculating their similarities, it is reasonable to observe the slight loss of speedups when 256 or 512 processors are utilise.

In order to prove that asynchronous migration benefits to the scalability of SPEO$_{\text{HPC}_{\text{cpu}}}$, we also present and compare speedups of SPEO$_{\text{HPC}_{\text{cpu}}}$ with its synchronous variant (SPEO$_{\text{HPC}_{\text{cpu}}}$ using synchronous migration, denoted as SYNC-SPEO$_{\text{HPC}_{\text{cpu}}}$) at Figure 4.3. Three main observations are listed:

- When a small number of CPU cores are utilised (less than 128), SYNC-SPEO$_{\text{HPC}_{\text{cpu}}}$ performs similarly with SPEO$_{\text{HPC}_{\text{cpu}}}$ for all dimension. It is because asynchronous migration is designed for the communication-intensive scenario with numerous processors. Thus the improvement could be slight if a small number of processors are utilise.

- Given the problem dimension, increasing CPU cores from 128 to 512 enlarges the gap between SYNC-SPEO$_{\text{HPC}_{\text{cpu}}}$ and SPEO$_{\text{HPC}_{\text{cpu}}}$. It demonstrates the benefits of asyn-

Table 4.2: Mean FEVs of diversity preserving and 3 simple buffer managements. Summarised statistical tests (+/≈/-) indicate basic buffers perform significantly better (+), worse (-), or similarly (≈) than the diversity preserving buffer.

$func$	Diversity Persevering	Best Persevering	Random Selection	FIFO
f_{23}	285.848	300.503 (-)	305.645 (-)	282.678 (≈)
f_{24}	112.325	113.48 (-)	114.839 (-)	117.438 (-)
f_{25}	121.458	124.631 (-)	128.128 (-)	127.552 (-)
f_{26}	100.057	100.061 (≈)	100.055 (≈)	100.058 (≈)
f_{27}	2.829	2.615 (≈)	2.385 (+)	2.495 (+)
f_{28}	370.483	367.893 (≈)	370.552 (-)	378.118 (-)
f_{29}	173.332	192.779 (-)	202.526 (-)	208.062 (-)
f_{30}	455.594	518.405 (-)	517.171 (-)	554.674 (-)
$+/\approx/-$	-	0/3/5	1/1/6	1/2/5

chronous migration when facing heavy communication workload with numerous processors.

- Given the number of CPU cores, increasing the dimension from 10 to 50 decreases the speedups of SYNC-SPEO$_{\text{HPC}_{\text{cpu}}}$ but has a small impact on the proposed SPEO$_{\text{HPC}_{\text{cpu}}}$. For example, SYNC-SPEO$_{\text{HPC}_{\text{cpu}}}$ achieves speedups of 447 when $D = 10$ and it decreases to 374 when $D = 50$. It is due to the fact that the increasing dimension enlarges the size of single message; as a result, the communication congest happens more frequently and seriously for SYNC-SPEO$_{\text{HPC}_{\text{cpu}}}$. Regarding the proposed SPEO$_{\text{HPC}_{\text{cpu}}}$, the increasing communication workload is hidden by asynchronous migration and thus similar speedups are achieved.

In conclusion, the asynchronous migration improves the scalability of SPEO$_{\text{HPC}_{\text{cpu}}}$ with better computational efficiency especially utilizing a large number of CPU cores or problems with a larger scale.

4.3.4 Performance Analysis on Diversity Preserving Buffer

In order to demonstrate the benefits of proposed diversity preserving buffer, we compare its performance with three basic buffers that are described as follows:

- Best preserving: compares the incoming immigrant with the worst existing immigrant in the buffer and stores the better one.
- Random selection: replaces a randomly selected existing immigrant in the buffer by the incoming immigrant.
- First-in-first-out (FIFO): inserts the incoming immigrant at the tail of the buffer and discards the first existing immigrant in the buffer.

Table 4.2 shows the solution quality of $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ with diversity preserving and three basic buffers on 8 test problems for $D = 10$. The aggregated statistic results at Table 4.2 prove that the diversity preserving buffer improves the solution quality of $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ compared to some basic buffers.

In order to figure out whether the diversity preserving buffer improve the solution diversity as expected, we compare the diversity of different buffers against the generations. The diversity is defined as the standard deviation of fitness values of all individuals [228] in the buffer. Here, we choose f_{28} for $D = 10$ as an example in Figure 4.4. Accordingly, diversity of all buffers increase rapidly from 0 at the beginning because many immigrants are filling in the empty buffer. Then, the diversity preserving buffer slightly decreases its diversity and keeps stable at a good level (≈ 600), while three basic buffers rapidly decrease the diversity to ≈ 100 and keep stable. Therefore, the diversity preserving buffer does improve the diversity of immigrants.

4.3.5 Performance Analysis on Topology Density

In order to investigate the impacts of topology density on the solution quality of $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$, we compare the solution quality obtained by different connection rates R_c ranging from 0.1% to 100% at Table 4.3. The bold values in the table are statistically better than the normal values. The results indicate that small connection rates (e.g. 0.1%, 1% and 2%) result in

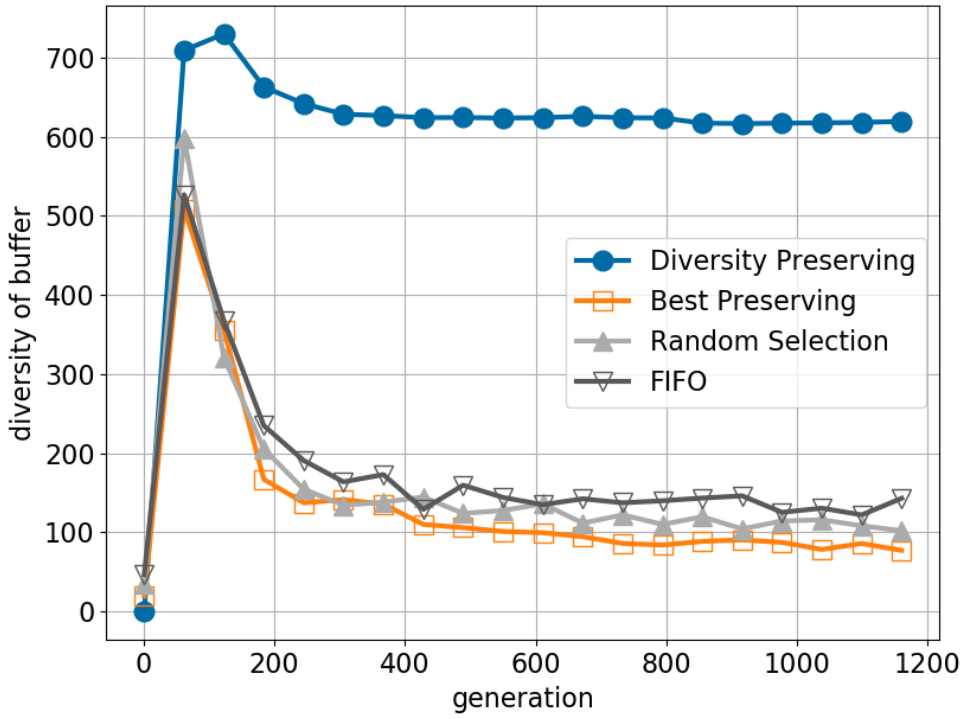


Figure 4.4: Example of diversity curve of different buffers on f_{28} for $D = 10$.

Table 4.3: Mean FEVs of $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ with connection rates $R_c = 0.1\%, 1\%, 2\%, 10\%, 25\%, 50\%$ and 100%.

func	$R_c = 0.1\%$[1]	$R_c = 1\%$	$R_c = 2\%$	$R_c = 10\%$	$R_c = 25\%$	$R_c = 50\%$	$R_c = 100\%$[2]
f_{23}	301.476	300.57	317.144	**268.488**	**285.848**	**288.14**	329.568
f_{24}	123.539	121.555	117.071	116.348	**112.325**	**113.614**	**114.482**
f_{25}	140.186	133.296	125.702	**122.165**	**121.458**	**123.26**	**121.66**
f_{26}	100.07	**100.062**	**100.051**	**100.058**	**100.057**	**100.054**	**100.054**
f_{27}	5.098	4.442	4.198	3.367	**2.829**	**3.039**	**2.748**
f_{28}	409.062	396.224	383.439	371.723	370.483	369.834	**365.262**
f_{29}	236.552	219.552	190.205	**176.168**	**173.332**	**167.395**	**171.741**
f_{30}	679.821	617.622	**468.777**	**436.164**	**455.594**	**441.438**	**457.314**

[1] $R_c = 0.1\%$ is equivalent to standard dynamic topology [163]
[2] $R_c = 100\%$ is equivalent to fully connected topology [66]

Table 4.4: Mean FEVs of $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ with improved dynamic and three common migration topologies for $D = 10$. Summarised statistical tests(+/≈/-) indicate common topologies perform significantly better (+), similarly (≈), or worse (-) than the improved dynamic topology.

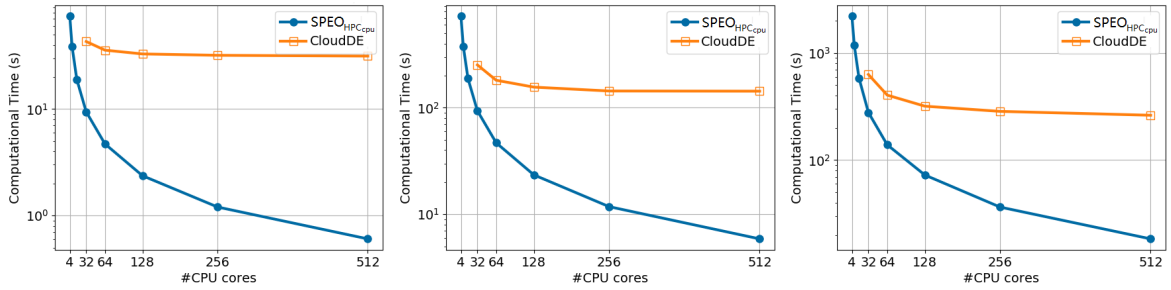
$func$	Improved Dynamic	Chain	Ring	Lattice
f_{23}	285.84	298.353 (-)	286.141 (-)	289.409 (-)
f_{24}	112.325	126.294 (-)	127.359 (-)	123.07 (-)
f_{25}	121.458	143.716 (-)	143.067 (-)	140.873 (-)
f_{26}	100.057	100.096 (-)	100.095 (-)	100.07 (-)
f_{27}	2.829	11.876 (-)	8.596 (-)	5.002 (-)
f_{28}	370.483	430.001 (-)	443.757 (-)	403.725 (-)
f_{29}	173.332	260.439 (-)	267.752 (-)	239.798 (-)
f_{30}	455.594	802.507 (-)	836.649 (-)	640.916 (-)
$+/\approx/-$	-	0/0/8	0/0/8	0/0/8

unsatisfactory solutions because the useful information is spread insufficiently among a large number of islands. Specifically, even total FEs is as large as 10^7 and the interval is 10, each island of $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ with a global population size $NP = 8192$ migrates only 120 times totally. As a result, if the connection rate is set as 0.1%, each island only migrates to at most 120 out of 512 islands within its life cycle which is far away from sufficiency. The results highly meet the existing findings which figured out the benefits of a dense topology. On the other hand, it is also unnecessary to set a very high connection rate which will result in a high communication cost but hardly improve the solution quality. In this case, a moderate connection rate around 25% is large enough.

In order to reinforce the above finding, Table 4.4 also compares the improved dynamic topology ($R_c = 25\%$) with three sparse common topologies that are chain, ring and lattice presented at Figure 2.1. It can be concluded that a dense topology indeed benefits to the solution quality because it completely outperforms three sparse topologies.

4.3.6 Performance Comparison with State-of-the-art Parallel EAs

The performance of $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ is further evaluated by comparing to CloudDE [36] that is a state-of-the-art parallel EA based on the island model and DE algorithm. In order to compare



(a) $D = 10$ (b) $D = 30$ (c) $D = 50$

Figure 4.5: Computational time of SPEO$_{\text{HPC}_{\text{cpu}}}$ and a state-of-the-art island-based parallel EA (CloudDE).

the capability and scalability of both algorithms with a large population over numerous cores, CloudDE configures the global population size and demands the number of CPU cores the same as SPEO$_{\text{HPC}_{\text{cpu}}}$. The rest configurations of CloudDE is chosen based on the default settings. Regarding the configurations of DE, CloudDE sets its four islands with different mutation schemes and CR values as follows:

- island 1: DE/rand/1/bin/$CR = 0.9$
- island 2: DE/rand/1/bin/$CR = 0.1$
- island 3: DE/best/1/bin/$CR = 0.9$
- island 4: DE/best/1/bin/$CR = 0.1$

In order to avoid the influence of DE algorithm on the comparison, SPEO$_{\text{HPC}_{\text{cpu}}}$ is set as the similar DE configurations as CloudDE. Specifically, i^{th} island is the same as island $mod(i, 4)+1$ of CloudDE where mod is the modulus operator; for example, 15^{th} island of SPEO$_{\text{HPC}_{\text{cpu}}}$ is set the same as island 4 of CloudDE. Regarding other model-related parameters, SPEO$_{\text{HPC}_{\text{cpu}}}$ employs default settings at Table 4.1.

Table 4.5 presents solution quality and statistical results on SPEO$_{\text{HPC}_{\text{cpu}}}$ and CloudDE for $D = 10$, $D = 30$ and $D = 50$. It can be observed that SPEO$_{\text{HPC}_{\text{cpu}}}$ performs similarly with CloudDE for $D = 10$ and significantly better than CloudDE for $D = 30$ and 50. It could

Table 4.5: Mean FEVs of $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ and state-of-the-art parallel EAs at $D = 10, 30$ and 50.

func	$D = 10$		$D = 30$		$D = 50$	
	CloudDE	$\text{SPEO}_{\text{HPC}_{\text{cpu}}}$	CloudDE	$\text{SPEO}_{\text{HPC}_{\text{cpu}}}$	CloudDE	$\text{SPEO}_{\text{HPC}_{\text{cpu}}}$
f_{23}	307.585	171.674 (+)	315.244	315.244 (≈)	344.005	344.005 (≈)
f_{24}	102.47	104.263 (≈)	222.727	223.033 (-)	261.648	255.39 (+)
f_{25}	106.623	106.821 (-)	204.124	203.34 (+)	211.132 (≈)	208.213 ≈)
f_{26}	100.084	100.056 (+)	100.198	100.172 (+)	100.309	100.232 (+)
f_{27}	1.236	1.87 (-)	403.233	400.477 (+)	502.719	488.194 (+)
f_{28}	352.456	356.838 (-)	748.942	776.991 (-)	1099.015	1149.6 (-)
f_{29}	250.72	173.222 (+)	1197.455	739.129 (+)	1449.592	866.763 (+)
f_{30}	474.846	467.22 (+)	1268.448	963.153 (+)	8536.38	8320.691 (+)
$+/\approx/-$	-	4/1/3	-	5/1/2	-	5/2/1

be due to the reason that CloudDE only employs 4 islands which can not provide sufficient searching diversity to find better solutions among an increasing number of local optima when the dimension increases.

We also compare the computational time of $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ with CloudDE at Figure 4.5 for 3 dimensions. It can be observed that $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ requires less computational time than CloudDE with the same computing resource, especially for a large number of processors. For example, when utilizing 32 CPU cores for Figure 4.5c, CloudDE requires more than double time of $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$; when CPU cores increase to 512, CloudDE requires 14.2 times of runtime than $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$. It is because CloudDE utilises synchronous communication scheme and it could be weakly scalable due to the increasing demands of global synchronisation between more and more cores.

4.4 Conclusions

This chapter proposes the $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ framework based on CPU-only HPC. This framework introduces an asynchronous migration. We then implement this framework with a standard DE algorithm and an improved dynamic topology for information exchange on up to 512 CPU

cores. The results present that $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ not only increases the computing efficiency but also improves the solution quality when compared to a state-of-the-art island-based parallel EA.

Chapter 5

Local Ensemble Surrogate Assisted Crowding DE and its Parallel Implementation based on the $\mathrm{SPEO}_{\mathrm{HPC}_{\mathrm{cpu}}}$ Framework

5.1 Introduction

When deploying the $\mathrm{SPEO}_{\mathrm{HPC}_{\mathrm{cpu}}}$ framework over many CPU cores, a large number of candidate solutions (e.g., 100,000,000 for $D = 100$ in Chapter 4) are now achievable. However, only a very small proportion of these solutions can facilitate the search process because EAs usually perform iterative generate-and-test operations. In other words, a very large number of solutions are directly discarded if they are evaluated as inferior, which wastes the most computing budget on unhelpful time-consuming fitness evaluation. However, any already evaluated candidate solutions, no matter superior or inferior, may carry some useful information about the search landscape in fact. Therefore, if the historical information that is carried by massive candidate solutions can be properly learnt and used, the search process by producing more superior candidate solutions can be facilitated.

One possible approach of using some of the already evaluated candidate solutions is to model the search landscape by surrogate models, which can estimate the quality of the newly generated candidate solutions. Therefore, we can generate more candidate individuals and filter the inferior ones by the trained model instead of evaluating their actual fitness values by time-consuming fitness evaluation. However, the training and utilisation of the surrogate model are very important but challenging because the performance of this kind of approach closely depends on the accuracy of the surrogate model. Specifically, if the search landscape is precisely modelled by sufficient data, the surrogate model can help to efficiently identify superior candidate solutions and accordingly facilitate search; otherwise, the solution quality even reduces when surrogate models are trained by insufficient data and only provide misleading information. From this perspective, the $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ can produce unprecedented data and thus assists in better modeling the problem landscape. It makes the $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ crucial for such surrogate approaches that require extensive learning data.

In this chapter, we design a local ensemble surrogate assisted crowding DE (LES-CDE). The proposed LSE-CDE builds many local surrogate models using extreme learning machine (ELM) and proposes a majority voting scheme to predict whether a trail vector is superior and worthy to be actually evaluated. After that, we implement the proposed LES-CDE in parallel based on the $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ (denoted as SPEO-LES-CDE). The SPEO-LES-CDE inherits the most features of the DE-based implementation of $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ except replacing the DE with CDE and introducing the ELM surrogate models. Experiments that compare the solution quality of original CDE and the sequential LES-CDE illustrate the superiority of the proposed LES-CDE over 8 test problems ($f_{23} - f_{30}$ of CEC2014) for three dimensions ($D = 10, 30$ and 50). Then we study the computing time that is required by sequential LES-CDE and CDE to achieve a small ($D * 10^4$) and large ($D * 10^6$) number of total FEs. We also present the computing time of SPEO-LES-CDE over up to 512 CPU cores in order to examine its computational efficiency. Results indicate that SPEO-LES-CDE only requires less than 12 minutes when using 50,000,000 total FEs to solve 50-dimension problems, while the CDE requires more than 2.5 hours and that of LES-CDE even exceeds the maximal execution time allowed by the HPC provider. Finally, we investigate the impacts of the chunk and volume of online training data

on the performance of SPEO-LES-CDE in terms of solution quality and computing speed.

The remaining chapter is organised as follows. Section 5.2 briefly introduces the background of ELM and its online learning version. Section 5.3 presents the proposed LES-CDE and its parallel implementation based on $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$. Experimental results are reported and discussed in Section 5.4. Finally, Section 5.5 draws conclusions.

5.2 Background

5.2.1 Extreme Learning Machine (ELM)

Extreme learning machine (ELM) [232, 233] is a highly time-effective single-hidden layer feedforward neural network (SLFN). In comparison with traditional methods, ELM has the outstanding performance in terms of speed and accuracy. ELM randomly initializes its hidden nodes, and calculate the output as follows: $f_L(\mathbf{x}) = \sum_{i=1}^{L} \beta_i h_i(\mathbf{x}) = h(\mathbf{x})\beta$, Where β_i is the output weight from the i-th hidden node to output node and $\beta = [\beta_1, \cdots, \beta_L]^T$. $h_i(\mathbf{x})$ is the output of i-th hidden node, $h(\mathbf{x}) = [h_1(\mathbf{x}) \cdots h_L(\mathbf{x})]$ is the output vector of input $\mathbf{x}$.

Bartlett's theory shows that feedforward neural networks perform better if achieving the smaller fitting error and smaller norm of weights. Thus, the objective is to minimize the fitting error: $min(\|\mathbf{H}\beta - \mathbf{T}\|^2)$ and the weight norm: $\|\beta\|$)

Where

$$\mathbf{H} = \begin{bmatrix} h(\mathbf{x}_1) \\ \vdots \\ h(\mathbf{x}_N) \end{bmatrix} = \begin{bmatrix} h_1(x_1) & \cdots & h_L(x_1) \\ \vdots & \ddots & \vdots \\ h_1(x_N) & \cdots & h_L(x_N) \end{bmatrix}$$

and $\mathbf{T} = [T_1, \cdots, T_N]^T$, where T_i is the real value of train data x_i.

The minimal norm least-square method is used to calculate $\beta = \mathbf{H}^{\dagger}\mathbf{T}$, where $\mathbf{H}^{\dagger}$ is the Moore−Penrose generalized inverse matrix of $\mathbf{H}$. According to the orthogonal projection method, $\mathbf{H}^{\dagger}$ and β is calculated as follows: $\mathbf{H}^{\dagger} = \mathbf{H}^T\left(\mathbf{H}\mathbf{H}^T\right)^{-1}$ and $\beta = \mathbf{H}^T\left(\mathbf{H}\mathbf{H}^T\right)^{-1}\mathbf{T}$.

5.2.2 Online Sequentially Extreme Learning Machine (OS-ELM):

When handling the data that arrive chunk-by-chunk, ELM has to be updated by online training the incoming data stream. Therefore, online sequential extreme learning machine (OS-ELM) was proposed in [234]. Let $(\mathbf{X}_i,\mathbf{T}_i)_{i=1}^{M_0}$ denotes the initially available input data. We first apply the original ELM to minimize $\|\mathbf{H}\beta - \mathbf{T}\|^2$. Then, after another chunk of data $(\mathbf{X}_i,\mathbf{T}_i)_{i=M_0+1}^{M_0+M}$ arrives, the problem turns into minimizing $\left\| \begin{bmatrix} \mathbf{H}_0 \\ \mathbf{H}_1 \end{bmatrix} \hat{\beta} - \begin{bmatrix} \mathbf{T}_0 \\ \mathbf{T}_1 \end{bmatrix} \right\|^2$ with $\hat{\beta} = \mathbf{K}^{-1} \begin{bmatrix} \mathbf{H}_0 \\ \mathbf{H}_1 \end{bmatrix}^T \begin{bmatrix} \mathbf{T}_0 \\ \mathbf{T}_1 \end{bmatrix}$ and $\mathbf{K} = \begin{bmatrix} \mathbf{H}_0 \\ \mathbf{H}_1 \end{bmatrix}^T \begin{bmatrix} \mathbf{H}_0 \\ \mathbf{H}_1 \end{bmatrix}$

5.3 The Proposed Method

5.3.1 LES-CDE Algorithm

LES-CDE uses historical search information to build multiple local surrogate models, and uses an ensemble of neighbouring local surrogates to guide the creation of promising trial vectors. The surrogate models employ the OS-ELM to build and online udpdate surrogate models. The algorithmic description of LES-CDE is presented in Algorithm 2. Several major components of LES-CDE are explained in details as follows.

5.3.1.1 Local surrogate model initialisation

To ensure the basic accuracy of local surrogate models, each model needs to be pre-trained before being used. A ratio r is defined to control the proportion of the total objective function evaluations ($maxFEs$) expended to generate pre-training data. When the current number of FEs is less than $r{\cdot}maxFEs$, the original CDE is executed with each of its generated trial vectors truly evaluated and stored in a training set $\mathbf{S} = \{\mathbf{s}_1, \cdots, \mathbf{s}_m\}$, where $m = \lfloor r * maxFEs \rfloor$. After this step, we obtain a population $\mathbf{P}_G = \{\mathbf{x}_{1,G}, \cdots \mathbf{x}_{NP,G}\}$ at generation G, which also serves as landmarks of the NP models. To build local models, each train data $\mathbf{s}_i$ is assigned to k sub-training sets of k models belonging to $\mathbf{s}_i$'s nearest k models represented by landmarks of $\mathbf{P}_G$. Then, ELM is used to initilise all local models denoted by $\mathbf{M} = \{\mathbf{m}_1, \cdots, \mathbf{m}_{NP}\}$.

Algorithm 2: The LES-CDE Algorithm

input : $\{NP, CR, F, k, nt, r = 0.3, L = 1000$, activation function is *sigmoidal*$\}$

Initialise $\mathbf{P}_0 = \{\mathbf{x}_{1,0}, \cdots, \mathbf{x}_{NP,0}\}$, $\mathbf{M} = \{\mathbf{m}_1, \cdots, \mathbf{m}_{NP}\}$, $\{\mathbf{T}_1 = \emptyset, \cdots, \mathbf{T}_{NP} = \emptyset\}$;
while *the predefined termination criteria is not met* **do**
  if $currFEs < r \cdot maxFEs$ **then**
    for $i = 1 \rightarrow NP$ **do**
      Generate trial vector $\mathbf{u}_{i,G}$ by $DE/rand/1/bin$ and evaluate fitness $f(\mathbf{u}_{i,G})$;
      Find $\mathbf{u}_{i,G}$'s nearest population member $\mathbf{x}_{j,G}$;
      Choose the better one from $\mathbf{x}_{j,G}$ and $\mathbf{u}_{i,G}$ as $\mathbf{x}_{j,G+1}$;
    end
  else
    if *Models* $\mathbf{M}$ *are not trained* **then**
      foreach $\mathbf{s} \in \mathbf{S}$ **do**
        Find k nearest population members $\{\mathbf{x}_{h_1,G}, \cdot, \mathbf{x}_{h_k,G}\}$ with respect to $\mathbf{s}$;
        Insert $\mathbf{s}$ into k training set $\mathbf{T}_{h_1}, \cdot, \mathbf{T}_{h_k}$;
      end
      for $i = 1 \rightarrow NP$ **do**
        Train $\mathbf{m}_i$ with train data $\mathbf{T}_i$ using ELM;
      end
    end
    $\mathbf{T}_1 = \emptyset, \cdots, \mathbf{T}_{NP} = \emptyset$;
    for $i = 1 \rightarrow NP$ **do**
      for $j = 1 \rightarrow nt$ **do**
        Generate $\mathbf{cu}_j$ based on the strategy $DE/rand/1/bin$;
        Find k nearest population members $\{\mathbf{x}_{h_1,G}, \cdot, \mathbf{x}_{h_k,G}\}$ for $\mathbf{cu}_j$;
        Estimate fitness of $\mathbf{cu}_j$ using k belonging models;
        if *more than* $k/2$ *estimated fitness are less than* $f(\mathbf{x}_i)$ **then**
          Set $\mathbf{u}_{i,G} = \mathbf{cu}_j$ and end this loop;
        else
          Record $\mathbf{cu}_j$ and the average of its nt estimated function values;
        end
      end
      if $\mathbf{u}_{i,G} = \emptyset$ **then**
        Set $\mathbf{u}_{i,G}$ as $\mathbf{cu}_j$ that has the smallest average estimated fitness from $\{\mathbf{cu}_1, \cdots, \mathbf{cu}_{nt}\}$;
      end
      Evaluate $\mathbf{u}_{i,G}$ to obtain its objective fitness value $f(\mathbf{u}_{i,G})$;
      Find k nearest population members $\{\mathbf{x}_{h_1,G}, \cdot, \mathbf{x}_{h_k,G}\}$ with respect to $\mathbf{u}_{i,G}$;
      Add $\mathbf{u}_{i,G}$ into k training set $\mathbf{T}_{h_1}, \cdot, \mathbf{T}_{h_k}$;
      Find $\mathbf{u}_{i,G}$'s nearest population member $\mathbf{x}_{j,G}$;
      Choose the better one from $\mathbf{x}_{j,G}$ and $\mathbf{u}_{i,G}$ as $\mathbf{x}_{j,G+1}$;
      $currFEs = currFEs + 1$, $G = G + 1$;
    end
    for $i = 1 \rightarrow NP$ **do**
      if $\mathbf{T}_i$ *is not* $\emptyset$ **then**
        Update $\mathbf{m}_i$ with $\mathbf{T}_i$ using OS-ELM;
      end
    end
  end
end

5.3.1.2 Local ensemble surrogate assisted trial vector generation

At generation G, LES-CDE sequentially generates nt trail vectors for each target vector. For each generated trial vector $\mathbf{cu}_j, j = 1, \cdots, nt$, its nearest k models will be identified. The estimated fitness value $\mathbf{cu}_j$ will be estimated by each of these k models, and then compared to the true fitness value $f(\mathbf{u}_i)$ of the target vector $\mathbf{u}_i$. The majority voting scheme (line 21 to line 30 at Algorithm 2) is employed to predict whether this generated trail vector can outperform the target vector. If so, this trial vector gets truly evaluated and no more new trial vectors for this target vector will be produced. Otherwise, another new trial vector will be generated and the above steps will be repeated. If all of the generated trial vectors are predicted as not outperforming the target vector, the trial vector having the best average fitness value estimation gets truly evaluated.

5.3.1.3 Local surrogate model updating

Trail vectors $\mathbf{U}_G = \{\mathbf{u}_{1,G}, \cdots, \mathbf{u}_{NP,G}\}$ produced at generation G will be used to update local models. Firstly, all sub-training sets $\{\mathbf{T}_1, \cdots, \mathbf{T}_{NP}\}$ will be emptied. Then each trail vector $\mathbf{u}_{i,G}$ is assigned to its k training sets $\{\mathbf{T}_{h_1}, \cdots, \mathbf{T}_{h_k}\}$ belonging to $\mathbf{u}_{i,G}$'s k nearest models represented in terms of $\mathbf{x}_{h_1,G}, \cdots, \mathbf{x}_{h_k,G}$. After all trial vectors are assigned to their corresponding models, we can obtain the training set $\mathbf{T}_i$ of each local model $\mathbf{m}_i, i = 1, \cdots, NP$, and use it to update $\mathbf{m}_i$ in an online manner. Here, OS-ELM is used for online learning.

5.3.2 Parallel Implementation of LES-CDE based on the $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ Framework

Although ELM is a simpler and faster training method compared to some traditional training ways, it is still very time-consuming if a significant volume of data is used to train the models. In order to obtain high-accurate surrogate models by training them with massive evaluated individuals in a reasonable time, we expect to accelerate the LES-CDE by using the $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ framework proposed in Chapter 4. Compared to original $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$, the proposed parallel LES-CDE (SPEO-LES-CDE) replaces the standard DE with CDE and additionally introduces the surrogate models.

Algorithm 3: SPEO-LES-CDE on i^{th} CPU core, $i \in [1, M_{cpu}]$

input : N_s, C_b, R_m, I, and parameters of LES-CDE

Initialise the island, the models the same as LES-CDE, $\mathbf{B} = \emptyset$;
while *the predefined termination criteria is not met* **do**
  if $currFEs < r \cdot maxFEs$ **then**
    *Perform CDE on $\mathbf{P}_{i,G}$ to generate and evaluate $\mathbf{P}_{i,G+1}$;
    $currFEs = currFEs + N_s$;
    Store all evaluated individuals in the training buffer $\mathbf{B}$;
  else
    if *Models* $\mathbf{M}$ *are not trained* **then**
      *Train the models LES-CDE using the training data buffer $\mathbf{B}$;
      $\mathbf{B} = \emptyset$;
    end
    if $size(\mathbf{B}) > C_b$ **then**
      *Use all training data in buffer $\mathbf{B}$ to update N_s models based on OS-ELM;
      $\mathbf{B} = \emptyset$;
    end
    *Perform CDE and majority voting on $\mathbf{P}_{i,G}$ to generate $\mathbf{P}_{i,G+1}$ based on the surrogate models $\mathbf{M}$;
    Evaluate $\mathbf{P}_{i,G+1}$;
    $currFEs = currFEs + N_s$;
    *Insert all evaluated trial vectors $\mathbf{U}_G$ into the buffer $\mathbf{B}$;
  end
  `/* Perform the main body of asynchronous migration proposed in Algorithm 1 */`
  Select recipients and send emigrants to other CPU cores based on R_m and I;
  Receive immigrants from other islands and store them in the buffer $\mathbf{B}$;
  $G = G + 1$;
end
* represents the same procedure with sequential LES-CDE

In Chapter 4, the received immigrants are used to build the diversity preserving buffer which always imports the unique and promising individuals into the current island. However, SPEO-LES-CDE, which exchanges information between islands in another way, uses them to train or update ELM surrogate models. Specifically, the island on each CPU core uses N_s ELM surrogate models that is represented by N_s individuals in the island, where N_s is the number of individuals of each islands.

The brief introduction of SPEO-LES-CDE can be seen at Algorithm 3, from which we

can observe that it shares the same communication scheme with $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ and the similar optimisation / training procedure with LES-CDE. The main difference between LES-CDE and the SPEO-LES-CDE is the collection of training data. The training data of SPEO-LES-CDE includes immigrants from other islands and the evaluated solutions from the current island. In SPEO-LES-CDE, the buffer simply stores all training data instead of performing diversity preserving operation. Regarding the model update, SPEO-LES-CDE initilises the surrogate models when the FEs reach $r * maxFEs$, and it updates these models once the buffer is full (exceed the buffer capacity C_b). The buffer will be cleared when the training data is used.

5.4 Experiments

In this section, several experiments are conducted to examine the LES-CDE and SPEO-LES-CDE. Firstly, we evaluate the performance of the proposed sequential LES-CDE under different parameter settings (k and nt) and find their best configurations for LES-CDE. Then we indicate the superiority of LES-CDE by comparing it with the original CDE in terms of the solution quality. After that, we present the execution time required by CDE, sequential LES-CDE and SPEO-LES-CDE using a small ($D*10^4$) and large ($D*10^6$) number of total FEs. Experiments using massive FEs illustrate that SPEO-LES-CDE can run great faster than CDE and LES-CDE which is even unable to finish running in a reasonable time. Finally, we investigate the impacts of the chunk and volume of online training data on the performance of SPEO-LES-CDE in terms of solution quality and computing speed.

5.4.1 Experiments Setup

The computing platform and test problems are exactly the same as the DE-based $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ in Chapter 4. We test the proposed algorithm and CDE use the strategy "DE/rand/1/bin" to generate trial vectors, and employ the commonly suggested parameter settings of $NP = 64$, $CR = 0.9$, and $F = 0.5$ [47]. In order to investigate the impacts of significant parameters k and nt, we examine the LES-CDE under different parameter settings: $[k, nt] \in$ [1,3,5,7] x [5, 9]. Some less sensitive settings in each ELM-based local surrogate model, such as the number

Table 5.1: Performance comparison of LES-CDE with different nt and k using the Iman and Davenport test with the Hochberg post-hoc procedure over 8 test functions at dimension 10, 30 and 50, respectively.

		$D = 10$	$D = 30$	$D = 50$
$nt = 5$	$k = 1$	*		
	$k = 3$	*	*	*
	$k = 5$	*	*	*
	$k = 7$		*	*
$nt = 9$	$k = 1$			
	$k = 3$	*		
	$k = 5$			
	$k = 7$			

of hidden neurons and the type of activation function are fixed (see Algorithm 2) based on default OS-ELM [234]. Each of the algorithms is executed for 15 independent times on each test problem at each dimension.

5.4.2 Study on Parametric Sensitivity

We firstly investigate how k and nt impact the solution quality of LES-CDE. In this experiment, k is set from $\{1, 3, 5, 7\}$ and nt is set from $\{5, 9\}$. We apply the Iman and Davenport test [235, 236] to compare the FEVs of each run achieved by all the algorithms over 8 functions at $D = 10, 30$ and 50, respectively. In order to figure out whether there is a group of configurations performing significantly better than the rests, we employ the post-hoc procedures [235, 236].

Table 5.1 presents the results of LES-CDE with 8 different configurations. Here, those leading to the statistically significantly better performance (at the significance level 0.05) over others are denoted by *. An empty cell means that corresponding algorithm is statistically significantly worse than some other algorithms. According to the result, LES-CDE with $k = 3$ or 5 performs significantly better than other configurations. Here, too small k ($k = 1$) performs bad because majority voting fails to work without sufficient estimated fitness values. On the

Table 5.2: Comparisons of mean FEVs between standard CDE and LES-CDE using $D * 10^4$ total FEs on 8 test problems for $D = 10, 30$ and 50. Statistical tests (+/≈/-) indicate LES-CDE performs significantly better (+), similarly (≈), or worse (-) than CDE based on Wilcoxon rank-sum test over 15 independent runs.

	$D = 10$		$D = 30$		$D = 50$	
$func$	CDE	LES-CDE	CDE	LES-CDE	CDE	LES-CDE
f_{23}	329.457	329.457 (≈)	315.244	315.244 (≈)	344.004	344.004 (≈)
f_{24}	137.979	138.999 (≈)	225.128	224.313 (+)	303.400	303.333 (+)
f_{25}	162.647	158.560 (+)	207.998	206.050 (+)	237.538	238.300 (≈)
f_{26}	100.230	100.241 (≈)	100.517	100.499 (+)	100.703	100.702 (+)
f_{27}	13.336	15.350 (−)	521.226	532.919 (−)	1803.149	1816.806 (−)
f_{28}	411.901	403.938 (+)	1117.660	1117.245 (+)	1785.477	1763.400 (+)
f_{29}	190.513	190.083 (+)	374.784	328.361 (+)	10630.261	10296.892 (+)
f_{30}	634.628	626.782 (+)	1597.820	1620.783 (≈)	10306.145	10155.024 (+)
$+/\approx/-$	-	4/3/1	-	5/2/1	-	5/2/1

other hand, too large k ($k = 7$) brings some inaccurate estimated fitness values, which are calculated by some far away models and mislead the majority voting. Moreover, increasing the value of nt cannot further improve the performance of LES-CDE. Based on these observation, we choose one of the best performed LES-CDE configuration: $k = 3$ and $nt = 5$ for the following experiments.

5.4.3 Performance Comparison of Solution Quality with CDE

Table 5.2 reports the comparison results of the proposed LES-CDE with CDE. Here, CDE acts as the control algorithm and is compared by LES-CDE based on the Wilcoxon's signed rank test respect to each dimension. According to the results, the proposed LES-CDE outperforms CDE for each tested dimension. This demonstrates the superiority of local ensemble surrogate models on improving the solution quality.

Table 5.3: Average computational time (hh:mm:ss) required by sequential CDE, LES-CDE and SPEO-LES-CDE to solve 8 test problems for three dimensions ($D = 10, 30$ and 50). The execution time presented use a small ($D*10^4$) and large ($D*10^6$) number of FEs, respectively. The sequential CDE and LES-CDE are conducted on a single CPU core and the SPEO-LES-CDE is conducted on 128, 256 and 512 CPU cores. The cost for demanding 512 CPU cores to conduct the entire experiments (8 test problems with 15 runs) are presented at the brackets.

MaxFEs	Algorithm	#CPU cores	$D = 10$	$D = 30$	$D = 50$
$D * 10^4$	CDE	1	00:00:04	00:00:32	00:01:35
	LES-CDE	1	00:05:26	00:24:23	00:57:49
$D * 10^6$	CDE	1	00:06:34	00:53:37	02:32:49
	LES-CDE	1	09:12:27	46:12:32	–:–:–[1]
	SPEO-LES-CDE	128	00:04:23	00:19:57	00:43:28
		256	00:02:15	00:10:02	00:22:01
		512 (21.76 USD/hour)[2]	00:01:09 (50.0 USD)	00:05:10 (224.3 USD)	00:11:35 (504.1 USD)

[1] It requires more computing time than the maximal execution time supported by NCI.
[2] Based on the pricing policy of AWC EC2 C5 instance.

5.4.4 Performance Comparison of Computing Speed

Although Table 5.2 illustrates the superiority of the LES-CDE, it naturally requires a longer computing time than CDE because a significantly larger computational budget is required to train surrogate models. Therefore, it is necessary to figure out how LES-CDE performs in terms of the computing speed, which determines the capability of LES-CDE solving increasing complex problems using a large population and massive FEs.

Table 5.3 records the average computational time (hh:mm:ss) that is required by sequential CDE, LES-CDE and SPEO-LES-CDE to solve 8 test problems for three dimensions ($D = 10, 30$ and 50). When $D*10^4$ total FEs are used, regardless that LES-CDE is much slower than CDE, LES-CDE and CDE can still finish within a reasonable time. However, when $D*10^6$ total FEs are used, LES-CDE completely fails to finish the execution for $D = 50$ due to the limitation of the maximal execution time of NCI (48 hours). Therefore, we then present the computing time of SPEO-LES-CDE using such a large FEs over 128, 256 and 512 CPU cores. Here, the

SPEO-LES-CDE employs a large population size NP=#CPU cores*N_s (island size $N_s = 64$). Results illustrate that the SPEO-LES-CDE can achieve such a large FEs in a short time. For example, LES-CDE fails to solve problems for $D = 50$ and CDE requires more than 2.5 hours, while SPEO-LES-CDE using 512 CPU cores only requires 11 minutes to achieve 100,000,000 total FEs. It is because the massive training tasks are distributed over a large number of CPU cores and the communication scheme of $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ can guarantee the efficient information exchange.

Basically, when researchers execute sequential EAs like CDE and LES-CDE, they hardly pay attend to the cost of computing facilities. However, purchasing computing powers from HPC or cloud providers can be sometime costly. Therefore, it is necessary to investigate how much the SPEO-LES-CDE may cost if it runs on such platforms. Table 5.3 also presents the cost for demanding 512 cores to conduct the entire experiments which include totally 120 instances (8 test problems and 15 runs for each problem). Here, we refer the pricing policy of AWS EC2 C5 CPU instance, which charges 21.76 USD to demand 512 CPU cores for one hour. It can be observed that a large expense is necessary to achieve such remarkable speedups and the cost increases significantly when the problem dimension becomes larger. For example, to conduct 120 instances, totally 2.3 hours and 50.0 USD is required when dimension is 10, while it reaches 23.2 hours and 504.1 USD when problem dimension is 100.

5.4.5 Analysis on Chunk and Volume of Online Training Data

As an online learning neural network, the performance (solution quality and computational efficiency) of SPEO-LES-CDE is impacted by the chunk and volume of training data. Specifically, the data chunk (buffer capacity C_b) determines how much data is used to update the model each time, and data volume (migration rate R_m) determines how much data is exchanged among islands. Therefore, we compare the performance of SPEO-LES-CDE using different data chunks ($C_b = 64, 128$ and 256) and volumes ($R_m = 0.1, 0.5$ and 1.0).

Table 5.4 presents the solutions quality of SPEO-LES-CDE using different data chunks and volumes. Results indicate that different configurations do not bring significantly different solution qualities. However, different computing speeds are observed according to the Table 5.5.

Table 5.4: Comparisons of mean FEVs of SPEO-LES-CDE using different data chunk and volume. Statistical tests (+/≈/-) indicate SPEO-LES-CDE performs significantly better (+), similarly (≈), or worse (-) than basic configuration ($R_m = 0.1$ and $C_b = 64$) based on 15 independent runs.

	$C_b = 64$			$C_b = 128$			$C_b = 256$		
	$R_m = 0.1$	$R_m = 0.5$	$R_m = 1.0$	$R_m = 0.1$	$R_m = 0.5$	$R_m = 1.0$	$R_m = 0.1$	$R_m = 0.5$	$R_m = 1.0$
f_{23}	279.69	295.78(≈)	283.6(≈)	223.47(≈)	292.23(≈)	239.05(≈)	238.77(≈)	288.9(≈)	250.83(≈)
f_{24}	128.81	131.07(−)	130.07(≈)	130.01(≈)	129.45(≈)	130.62(−)	130.32(≈)	131.0(−)	129.71(≈)
f_{25}	146.65	146.3(≈)	147.5(≈)	147.45(≈)	148.48(≈)	146.27(≈)	148.37(≈)	146.53(≈)	147.31(≈)
f_{26}	100.18	100.15(+)	100.17(≈)	100.18(≈)	100.17(≈)	100.17(≈)	100.17(≈)	100.18(≈)	100.18(≈)
f_{27}	12.01	12.43(≈)	12.89(≈)	12.84(≈)	12.41(≈)	12.1(≈)	11.97(+)	12.15(≈)	12.88(≈)
f_{28}	401.5	412.39(≈)	402.88(−)	395.48(≈)	396.78(≈)	395.63(≈)	402.56(≈)	395.95(≈)	403.17(≈)
f_{29}	154.77	151.68(≈)	154.44(≈)	157.37(≈)	156.39(≈)	151.99(≈)	153.59(≈)	151.48(≈)	152.03(≈)
f_{30}	582.93	576.78(≈)	586.85(≈)	576.4(≈)	576.83(≈)	582.51(≈)	578.86(≈)	596.1(≈)	597.15(−)
$+/\approx/-$	-	1/7/1	0/7/1	0/8/0	0/8/0	0/7/1	1/7/0	0/7/1	0/7/1

Table 5.5: Average computational time (hh:mm:ss) that is required by SPEO-LES-CDE with different buffer capacities ($C_b = 64, 128$ and 256) and migration rates ($R_m = 0.1, 0.5$ and 1) to solve 8 test problems for $D = 10$. Totally $D * 10^6$ FEs are used herein over 512 CPU cores at CPU node at NCI HPC.

	$C_b = 64$	$C_b = 128$	$C_b = 256$
$R_m = 0.1$	00:01:09	00:01:20	00:01:36
$R_m = 0.5$	00:01:58	00:02:15	00:02:34
$R_m = 1.0$	00:03:05	00:03:28	00:04:07

Results indicate that the data volume R_m has significantly more impacts on the speed. For example, when C_b increases from 64 to 256, the computing time of SPEO-LES-CDE with $R_m = 1.0$ only increases from 3 minutes to 4 minutes. However, when R_m increases from 0.1 to 1.0, the computing time of SPEO-LES-CDE with $C_b = 256$ increases from 1.5 minutes to 4 minutes. The reason is that a large data volume not only increases the communication workload between CPU cores, but also increases the workload of training process. In summary, to balance the solution quality and computing speed, a small data chunk and volume are recommended for SPEO-LES-CDE.

5.5 Conclusions

This chapter proposes the LES-CDE which can uses historical search information to train multiple local surrogate models. Moreover, an ensemble of several neighbouring local models is applied to guide the generation of promising trial vectors. To reduce model training costs, ELM is used to build surrogate models and OS-ELM is used to update these models in an online manner. Furthermore, we also implement it in parallel based on the $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ framework. In experiments, we compared the original CDE, sequential LES-CDE and SPEO-LES-CDE in terms of solution quality and computing speed. Experiments indicate that the LES-CDE outperforms CDE in terms of the solution quality and the SPEO-LES-CDE can make up the slow computing speed of LES-CDE if massive FEs are used. However, we also figure out that a significant cost is necessary to achieve such a remarkable speedup. Finally, we provide a recommendation on configuring the chunk and volume of online training data to achieve both satisfactory solution quality and computing speed.

Chapter 6

Correctness Verification for Implementing Parallel EAs based on a Single GPU

6.1 Introduction

In Chapter 4 and Chapter 5, we have shown that a large number of CPU cores can remarkably speedup the time-consuming EAs if designed and executed in parallel. However, it also can be very costly for researchers to demand so many CPU cores; thus, modern powerful GPUs that require a lower cost and provide a larger computing power have entered computing's mainstream. So far, many EAs that were sequentially implemented on CPU are now redesigned and implemented based on GPU and achieve significant speedups [73–78]. However, compared to traditional serial-oriented programming, GPU-based programming is more complicated and thus obtaining correct outputs by GPU-based EAs is not so straightforward as the CPU-based EAs. The reasons include that the parallel programming brings in many challenges, which are not typically encountered in the conventional serial-oriented programming. For example, the memory barriers are essential in parallel computing for preventing the access conflicts when multiple threads write to the same shared data. Moreover, to program on GPUs requires

programmers to have grown skills and a certain level know-how of GPU hardware structures. For instance, the developer needs to understand the characters and differences between six types of GPU memories and uses them, properly. In addition, monitoring thousands of threads makes it difficult to debug the GPU programming. In summary, implementing GPU-based programs is very challenging and thus guaranteeing the correctness before applying these programs is necessary.

Many existing works [237–239] studied how to verify the correctness in some simple GPU-based applications. Some commercial software, e.g. Matlab GPU Coder^{TM}, even provides an API for checking the correctness of GPU codes before executing them on GPUs. Due to the stochastic nature of EAs, existing works, which focus on simple GPU-based applications, are not capable of verifying the correctness of complex GPU-based EAs. In other words, it still lacks the work that can verify the correctness of GPU-based EAs when they are migrated from CPU-based programming. Therefore, many GPU-based EAs [79–82] observed the biased outputs with their counterparts but failed to figure out the reasons. As a result, it exists risks that GPU-based EAs may output incorrectly even they can provide remarkable speedups.

Instead of studying accelerating parallel EAs based on a single GPU, this chapter propose guidelines for EA researchers to verify the correctness after they implement EAs based on a single GPU. In this chapter, an example of migrating the PSO from CPU based coding to the GPU environment is firstly given to show why correctness verification is necessary for GPU-based EAs. Then, this chapter discusses some GPU-inherent issues including the library functions, the numerical precision, and the race condition which influence the output of GPU-based EAs. To cope with the issues mentioned above, a set of guidance is proposed to verify the correctness of the GPU-based EAs. Finally, the effectiveness of the proposed guidelines is examined by employing a working example based on a GPU-based modified brain storm optimisation (MBSO) [121, 240].

The rest of the chapter is organised as follows. Section 6.2 presents an example revealing the difficulty and the necessity of correctness verification for GPU-based PSO. Moreover, the impact on the outputs caused by the GPU-based EAs is discussed from aspects of three GPU-inherent issues including the library function, the numerical precision, and the race condition.

Section 6.3 proposes a set of guidance for verifying the correctness of the GPU-based EAs against the issues mentioned above. Section 6.4 presents a working example based on GPU-based MBSO to demonstrate the effectiveness of the proposed guidelines.

6.2 Issues and Analysis

The correctness verification did not attract much attention in existing works when most EAs were designed for running on the conventional serial-based computing facilities because the correctness verification can be simply achieved by comparing the outputs of the different implementations of the same algorithm, directly. However, things get complicated when the EAs are put on the GPU-based environment because the difference of the outputs generally caused by the unpredictable execution order of the parallel processes. An example of the conventional PSO algorithm is given as an example in both the sequential and the GPU-based ways in this section to showcase the difficulty and failure of correctness verification in the GPU-based environment against the CPU-based environment. In addition, four programming languages including Matlab, Python, C/C++, and CUDA are used to establish four versions of PSO to get more objective results. Four test functions, namely, Sphere, Ackley, Griewank, and Rastrigin, are used with $D = 10$ where D is the problem dimension. The outputs of solution quality are measured by the mean FEVs [88] of 30 independent instances repeated with different random seeds. The configuration of these PSO implementations are based on the standard PSO [241], and the maximal fitness evaluation is set to $D * 10^4$ as the stopping criterion. It has been known in common sense that the Random Number Generators (RNGs) influence the outputs of the EAs. Thus, two different RNG configurations are used in the example and are listed as follows:

- *Employing the default RNG in the programming language*: The Mersenne Twister (MT) is used in Matlab and Python; the Linear Congruential Generator (LCG) is used in C/C++, and the Xorshift is used in CUDA.

- *Employing the static RNG file*: To eliminate the differences caused by different RNGs, a pregenerated file containing sufficient random sequences is fed as the input to ensure the

Table 6.1: Mean FEVs of four implementations of PSO with **different RNGs** on four functions.

	Sphere	Ackley	Griewank	Rastrigin
Matlab	0.0210 (0.019)	16.5337 (8.296)	0.1939 (0.108)	26.8214 (9.242)
Python	0.0155 (0.014)	12.3227 (9.116)	0.1475 (0.119)	29.3329 (8.667)
C/C++	0.0194 (0.018)	17.8694 (7.004)	0.2415 (0.180)	25.3294 (9.583)
CUDA	0.0190 (0.017)	16.7155 (8.064)	0.2152 (0.114)	33.3595 (9.022)

Table 6.2: Mean FEVs of four implementations of PSO with the **identical RNG** on four functions.

	Sphere	Ackley	Griewank	Rastrigin
Matlab	0.0119 (0.012)	17.8308 (7.173)	0.1393 (0.068)	28.5840 (8.495)
Python	0.0119 (0.012)	17.8308 (7.173)	0.1393 (0.068)	28.5840 (8.495)
C/C++	0.0119 (0.012)	17.8308 (7.173)	0.1393 (0.068)	28.5840 (8.495)
CUDA	0.0326 (0.024)	15.3420 (8.831)	0.2222 (0.138)	26.4825 (9.287)

identical random numbers are used in all implementations.

The mean FEVs and the standard deviations (in brackets) of all test functions and implementations are shown in Table 6.1 (default RNGs) and Table 6.2 (unified RNG). According to Table 6.1, it is observable that the same sequential PSO algorithm implemented in different programming languages present unidentical results and bring in ununified searching behaviours to PSO when the default RNGs are used in the experiment. On the contrary, all implementations present same searching behaviours and results in Table 6.2 after employing the same random sequence file except the CUDA-based implementation. To conclude in short, the research topic of how to verify the correctness was not raised in the earlier researches because the correctness can be easily verified utilizing the unified RNG. It is an interesting phenomenon that the results still different even the random sequence is identical to the others. In this scenario, neglecting the unexpected biases without verifying the correctness making the outcome of GPU-based EAs unreliable because researchers will have no idea about whether the outcome of the GPU-based EAs is correct or not. Motivated by the phenomenon revealed in the example, some GPU-inherent issues that may influence the outputs of the GPU-based EAs

are studied and a set of guidance to verify the correctness of GPU-based EAs is proposed.

In this chapter, we investigate three fundamental and GPU-inherent issues, which cause the biases to EA results. Related discussions remain vacant in the literature of the existing GPU-based EAs. According to the observation on the example given in the previous section, we've known that the default RNGs in different programming language produce unidentical random sequences and cause the biased results in EAs. However, there are other elements that influence on the resulting consistency. The libraries and functions built by different programmer perform different but all correct results when the feasible answer is not unique. Another thing caught our appetite is the numerical precision because the EAs naturally involve a lot of numerical operations in the processes. The other issue is the commonly seen race conditions in the parallel computing system.

6.2.1 Build-in Functions and Libraries

Many of the build-in functions can be found in all well-developed programming languages providing convenient and efficient assistance to the developers to reduce the programming burden. These functions are customisedd for a specific programming language to maximise the execution performance. As a result, functions built for the same purpose may produce different results in different programming languages or platforms even if the identical inputs are given. Taking the same example from the previous section, all RNGs in different programming languages are designed to generate random numbers, but the outputs are entirely different even if the same random seed is used.

Comparing the libraries for CPU and GPU, we can say that these two libraries are dissimilar because the GPU libraries are designed to include many parallel processes and implemented for maximally utilse thousands of parallel threads while CPU libraries, in general, designed to support much fewer threads to be executed in parallel. Therefore, GPU libraries can be difficult for average programmers to understand. Some of the GPU libraries are even closed source without revealing the process of manipulating data. As a result, the detail of functions in GPU libraries are not generally known, and the biases caused by the functions and libraries in GPU are easily neglected. To verify the correctness of the GPU-based EAs, solutions to

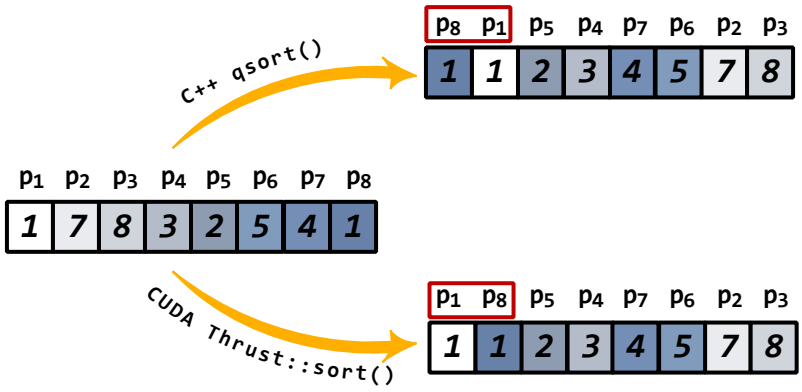


Figure 6.1: Different outcomes of sorting function by C++ and CUDA.

these issues need to be developed.

Example: Besides the example of random number generation given above, the fundamental functions in the build-in libraries provide efficient support to the programmers. Nevertheless, many of them have the same issue of producing correct but different results. Taking the sorting function, which is heavily used in the rank-based fitness assignment [242, 243], as an example, the sorting is used in EAs when calculating the fitness values based on the ranking of the objective values of all individuals. The left side of figure 6.1 shows an example containing eight individuals $\{p_1, \ldots, p_8\}$ with objective values $\{1, 7, 8, 3, 2, 5, 4, 1\}$ in the queue.

In this example, qsort() function in the C++ library and Thrust::sort() [244] function in the CUDA library are employed. The right side of figure 6.1 shows the results obtained by C++ and CUDA build-in functions after sorting. The sorting results are $\{p_8, p_1, p_5, p_4, p_7, p_6, p_2, p_3\}$ and $\{p_1, p_8, p_5, p_4, p_7, p_6, p_2, p_3\}$, which are obtained by qsort() and Thrust::sort(), respectively. It is noticeable that these functions produce different sorted indices for p_1 and p_8 because they are implemented with different sorting algorithms. The qsort() in C++ library is built based on the quick sort algorithm [245], which is unstable when more than one elements contain the identical value in the list. On the other hand, The Thrust::sort() is implemented based on the merge sort [246] and the radix sort [247] algorithms, which are all stable sorting algorithms regardless whether repeated values are included in the sorting list. In other words, having two elements (p_1 and p_8) with the same value in this example, qsort() may return the sorted result

in different orders, but Thrust::sort() will always keep the result identical. As a result, the bias caused by the sorting would eventually grow in operations such as the parent selection in EAs and produce different outcomes.

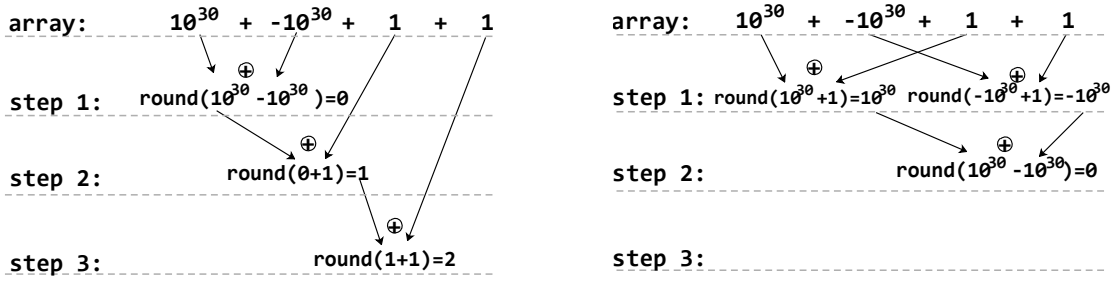
6.2.2 Numerical Precision of Floating Point

In most cases of numerical computation on computers, the real numbers are represented by floating point. Since floating point arithmetic is an approximation way, there are usually some loss of precision when presenting a real number by floating point. Especially, when the number belongs to the irrational number category. Therefore, in some crucial situations such as scientific computing, numerical precision is an important issue that influences the accuracy of results and needs to be carefully studied. To have a common standard for coping with this issue, IEEE-754 standard [248] is introduced to ensure the reliability and portability of floating point across different software or hardware platforms. NVIDIA also follows this standard to product all generations of its GPU products. The current generation of NVIDIA GPUs such as Tesla P100 and GTX1080 support both single and double precision defined by IEEE-754 standard for most of the CUDA-based numerical computations. However, in order to achieve better performance, there could be a trade off between numerical precision and computing speeds for GPU in some cases, which results in the biases between CPU and GPU on the results of numerical computation.

EAs usually consist of various numerical computation operators such as summation and multiplication. As the matter of fact, CPU and GPU show different precision on these operations and do result in the biases, which influence the outputs of EAs. Although the bias from a single operation is relatively small, the effect can be accumulated over thousands of generations. Therefore, the numerical precision plays an important role, which significantly affects the computing results and increases the difficulty of correctness verification.

Example: An example of numerical precision in EAs is the roulette wheel selection [117] which calculates the selecting probability of parent $\mathbf{x}_i$ as follows:

$$P(\mathbf{x}_i) = \frac{f(\mathbf{x}_i)}{\sum_{i=j}^{NP} f(\mathbf{x}_j)}$$



(a) sequential summation

(b) parallel summation reduction

Figure 6.2: Example of arithmetic operation associations.

where $f(\mathbf{x}_i)$ is the fitness value of $\mathbf{x}_i$ and NP is the population. When implementing the summation of all the fitness values ($\sum_{j=1}^{NP} f(\mathbf{x}_j)$) for roulette wheel selection on GPU, parallel reduction, which reduces the time complexity form $O(n)$ to $O(log(n))$ by adding pairwise data elements in parallel, is usually employed. In order to make the values significant and easy to compare, fitness values of four chromosomes in this example are set as $\{10^{30}, -10^{30}, 1, 1\}$. The traditional CPU-based sequential summation and GPU-based parallel reduction are shown as follows:

- CPU: a sequential loop is employed to sum all elements. Figure 6.2a shows the summation value $\sum_{i=1}^{NP} f(\mathbf{x}_i) = 2$.
- GPU: Parallel reduction executed on GPU is shown at Figure 6.2b. The summation value $\sum_{i=1}^{NP} f(\mathbf{x}_i) = 0$.

Sequential summation and parallel reduction output with different precision because they have different association of arithmetic operation. As a result, *round*() function rounds different real values that are summed by different pair of numbers which produces the different losses of precision when converting a real value to a floating point. In conclusion, the issue of numerical precision such as association of arithmetic operation can bring different numerical precision as well as the output of EAs.

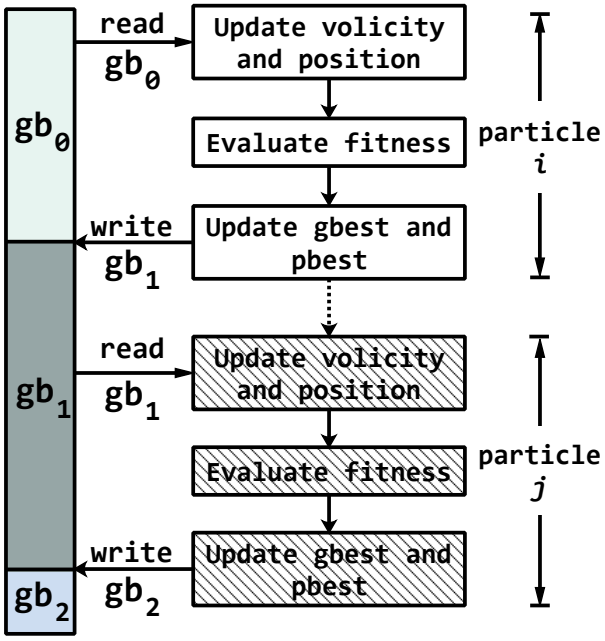


Figure 6.3: Runtime flowchart of asynchronous sequential PSO based on CPU.

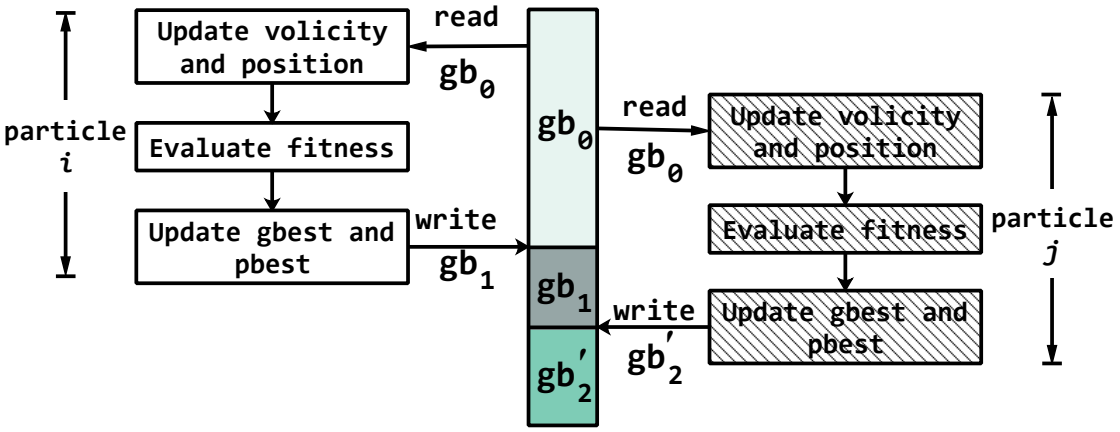


Figure 6.4: Runtime flowchart of asynchronous parallel PSO based on GPU.

6.2.3 Race Condition

When GPU-based parallel programs are executed on GPU, computing tasks are assigned to different threads to process concurrently. GPU groups threads in warps and dispatches them

based on runtime status of GPUs and thus thousands of GPU threads are executed disorderly. Consequently, if there is any interaction between more than two threads, race condition [249] occurs and outputs are uncertain; otherwise, a parallel program on thousand threads perform the same with a sequential program on a single thread.

Race condition makes correctness verification of GPU-based EAs more challenging due to their unexpected outputs. The main source of race condition in GPU-based EAs is population updating mechanism. Namely, when EAs are implemented on GPU, the entire population are usually processed by many threads in parallel. Since each individual may read from and write to other individuals for crossover and selection operations, the different accesses orders may lead to different outputs.

Example: An example is given to show how asynchronous PSO (APSO) [250, 251] triggers race condition when it is executed in parallel. Figure 6.3 shows the sketch of sequential APSO in which particle i, j read and update global best by turns. Namely, particle i firstly reads the global best $\mathbf{gb}_0$ and updates it as $\mathbf{gb}_1$, then particle j reads global best $\mathbf{gb}_1$ and updates it as $\mathbf{gb}_2$. Regarding GPU-based APSO, there is no guarantee that particle j will wait for the finish of particle i. Figure 6.4 presents a possible case that particle j reads $\mathbf{gb}_0$ before particle i updating the global best. In this example, the new position of particle j that is generated based on $\mathbf{gb}_0$ is different with that of sequential APSO in which particle j is generated based on $\mathbf{gb}_1$. Moreover, global best of GPU-based parallel APSO and sequential APSO are $\mathbf{gb}'_2$ and $\mathbf{gb}_2$, respectively. As a result, the final outputs of GPU-based and CPU-based PSO must be different after thousands of generations. Moreover, the issue can be more significant if the population size is large because there are A_{NP}^{NP} possible process orders for a population with NP individuals.

6.3 The Proposed Guidelines

This section describes a set of guidelines to verify GPU-based EAs considering the context and issues that are studied in previous section. Here, we give a short overview of the guidelines and then describe the details.

Figure 6.5 show the four main steps for correctness verification. Firstly, we obtain the

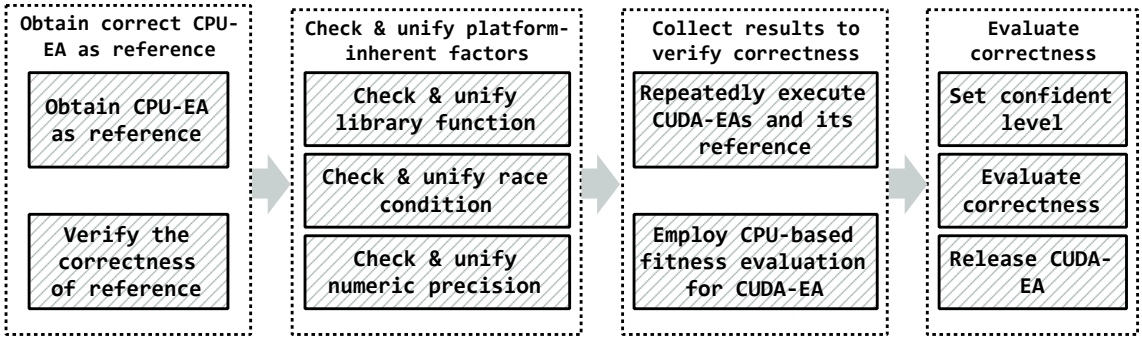


Figure 6.5: Guidelines of correctness verification of GPU-based EAs.

CPU-based sequential implementation which works as reference. Then we address the GPU-inherent issues to avoid their influence on correctness verification. Thirdly, we run the tests and compare optimisation accuracy collected from GPU-based EA and its CPU-based reference. Finally, we evaluate and confirm the correctness based on the comparison results.

6.3.1 Obtaining Correct CPU-based EAs as the Reference

Obtaining a correct reference of GPU-based EAs is the foundation of correctness verification. This first step mainly focuses on developing a correct reference.

Obtaining a CPU-based EAs as the reference: A CPU-based sequential implementation of EA is required as the reference for correctness verification. If this reference does not exist, developers are suggested to implement it with CPU-based programming languages such as Matlab or C/C++.

Confirming the correctness of the reference: When a sequential implementation of EA is obtained as the reference, its correctness needs to be carefully confirmed. Since developers usually have much experience in implementing sequential programs, correctness verification of the reference is not a difficult task.

6.3.2 Unifying GPU-inherent Issues

This step focuses on avoiding the biases that are caused by GPU-inherent issues when comparing a GPU-based EA with its reference. Guidelines of checking and unifying three issues

(library functions, numerical precision and race condition) are provided as follows.

Unifying library functions: The unification of library functions can be divided into two aspects. For some library functions that output completely differently with CPU libraries such as RNGs, we can unify them as follows:

- Transfer the input date from GPU global memory to CPU host memory
- Employ corresponding CPU library functions to calculate the output with the input data
- Transfer the output data back to GPU global memory

Regarding some library functions that occasionally bring small bias, the necessity of replacing them with CPU library functions is up to developers' expertise on GPU-based programming. Although the replacement is not compulsory, it increases the credibility of final conclusion for the correctness verification.

Unifying numerical precision: Since the association of arithmetic operations is the most common reason that causes the issue of numerical precision, developers can check whether there are mathematical operators like $\sum$ or $\prod$ in GPU-based EAs. If this operator exists and is implemented with the parallel reduction method, developers can unify by replacing it with a sequential way. Similar with some library functions, unification of numerical precision is also not compulsory because the bias caused by numerical precision are not large in most cases.

Avoiding race condition: In order to check the issue of race condition, developers can examine the existence of the shared variable in GPU-based program. If a variable that is stored in GPU global memory is accessed by two or more threads (at least one thread writes data to this shared data), race condition occurs probably. Especially, the population updating mechanism needs to be examined carefully because race condition probably exists if asynchronous mechanism is employed. To avoid the race condition in this case, it is recommended to replace the asynchronous population updating mechanism with synchronous one for both GPU-based EA and its reference.

6.3.3 Collecting Results

Employing CPU-based fitness evaluation for GPU-based EAs: Our work aims to verify the correctness of the algorithmic implementation of GPU-based EAs excluding the implementation of test problems. In order to avoid the uncertainty brought by fitness evaluation during correctness verification, we suggest to utilse the CPU-based implementation of fitness evaluation for both the GPU-based EA and its reference. In this way, GPU-based EAs can transfer the entire population to CPU memory for fitness evaluation and then transfers them back to GPU global memory for further processes.

Executing GPU-based EAs and the reference repeatedly: In order to comprehensively evaluate correctness of GPU-based EAs and avoid the influence of occasional issues, multiple runs with various configurations for GPU-based EA and its reference are suggested. The configurations could be random seeds, algorithmic parameters, test problems and so on.

6.3.4 Evaluating Correctness

Setting tolerance rate: According to above discussion, some GPU-inherent issues occasionally produce small biases but take lots of efforts to unify, thus their unification are not compulsory and up to developers' expertise on GPU-based programming. Therefore, the tolerance of noises that caused by non-bug issues is necessary to avoid the misjudgment of some correct instances. Here, tolerance rate $T\%$ is designed to represent how reliable the conclusion of correctness verification is. A higher T makes the confirmation of correctness more reliable but it may fail to verify an indeed correct GPU-based EAs because it is sensitive with the noise caused by above non-bug factors. On the contrary, a lower T tolerates many occasional issues but reduces the reliability of correctness verification. A high T is suggested if:

- The unification is comprehensively employed on most GPU-inherent factors
- The implementation of GPU-based EA is relatively simple and there is no complex operation that is difficult to code in parallel.
- The application requires a highly reliable confirmation of correctness of GPU-based EAs.

- A large number of instances with different configurations or problems are tested .

The tolerance rate T works as a threshold that represents the minimal ratio of correct instances among all tested instances. Specifically, the correctness of a GPU-based EA is confirmed if it outputs correctly for more than $n * T\%$ out of n instances, where n is the number of total instances.

Evaluating correctness: Before comparing the optimisation accuracy of GPU-based EA with that of its reference, users are required to specify the acceptable error level (AEL) which relies on each test problem. AEL represents the maximal systemic error for a specific problem and any bias that is smaller than AEL is neglected in correctness verification. For example, if our test problems are selected from CEC2014 benchmark, we can set AEL at 10^{-8} because a solution is regarded as the global optima when its FEV is less than 10^{-8} [88].

Once AEL is specified, the correctness of the GPU-based EA is evaluated by comparing its optimisation accuracy with that of its reference for each independent run. Specifically, if the bias is smaller than AEL, GPU-based EA is counted as correct at this run. Then the comparison results for all runs are synthesised and users can confirm the correctness if the ratio of correct runs are larger than $T\%$.

Releasing GPU-based EAs: If the correctness of the GPU-based EA is verified, all the unification can be discarded and the original implementation is ready to release because all unification are only designed for correctness verification and may reduce the computational efficiency.

6.4 A Working Example: Implement and Verify GPU-based MBSO

In order to examine the effectiveness of guidelines, we implement a novel EAs called MBSO based on a single GPU and verify its correctness according to the above guidelines. Here, MBSO is selected as the working example because it is a state-of-the-art EA and have more complex evolutionary operations compared to some traditional simple EAs. The proposed

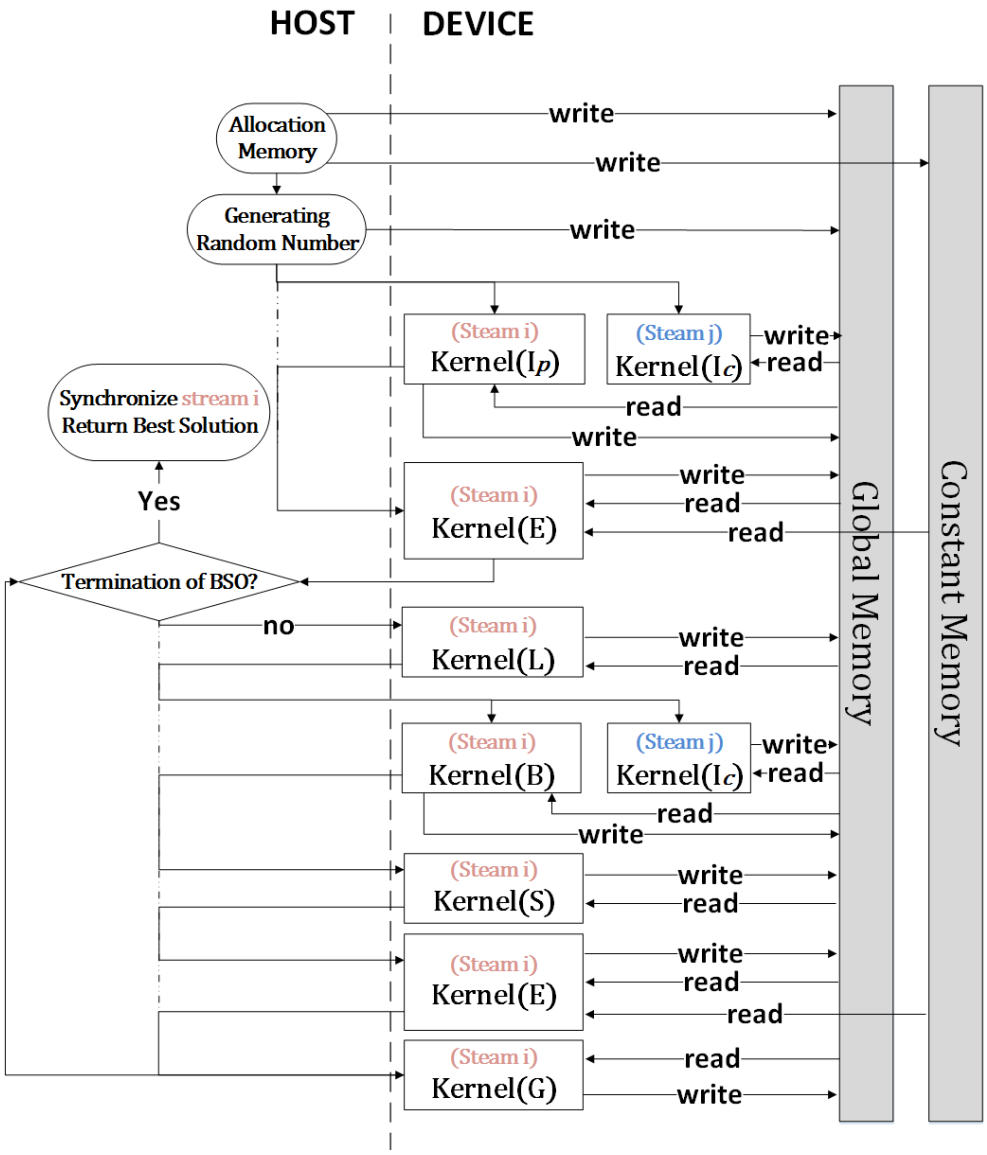


Figure 6.6: Flow chart of GPU-based MBSO.

GPU-based MBSO encapsulates all of the evolutionary operations into seven kernel functions that are illustrated in Figure 6.6.

6.4.1 Implementation of GPU-based MBSO

6.4.1.1 Initialization of ideas and clusters

kernel($\mathbf{I}_p$) initialise all the ideas randomly. This kernel uses one thread for each elements of the idea array and uses the global memory to store population data.

Once all of the ideas are generated, **kernel(E)** is employed to evaluate the quality of these ideas. This kernel calculates the objective function values of the ideas and writes them into the global memory. The objective function is defined according to the problem being solved and thus has varying complexity. Here, we employ CEC2014 benchmark functions as the optimization problems and implement them based on CUDA. Each thread of this kernel operates on one element of the idea array and thus this kernel has the same block size number as **kernel($\mathbf{I}_p$)**. Since the benchmark functions contain the addition and/or multiplication operations on all elements of a population member, we employed the parallel reduction to make the sum and production calculations. It is worth of noting that the objective function evaluation could be the most time-consuming, especially when the problem and population size grow. Therefore, the effective parallelization of this part on GPU may lead to remarkable computational speedup. In addition to the global memory, **kernel(E)** also uses the shared memory and the constant memory to store the data that needs to be frequently used.

Beside idea initialization, cluster centers are initialized by **kernel($\mathbf{I}_c$)** and written into the global memory. Since this kernel is independent of **kernel(E)**, it can be executed in a different streams to increase computational efficiency [17]. To ensure cluster centers are successfully initialized before being used by other kernels, two streams w.r.t. **kernel(E)** and **kernel($\mathbf{I}_c$)** need to be synchronized. Similar to **kernel($\mathbf{I}_p$)**, each thread of kernel(Ic) operates on one element of the idea array and thus **kernel($\mathbf{I}_c$)** has the same block size as **kernel($\mathbf{I}_p$)**.

6.4.1.2 Convergent operator

Convergent operator involves the calculation of distances between all ideas and group (cluster) centers, which is suitable for be parallelized on GPU. **kernel(L)** calculates the distances between one idea to every group center and labels this idea with the index of its nearest group

center. In this kernel, M group centers initialized by **kernel($\mathbf{I}_c$)** are firstly loaded into the shared memory to avoid frequent global memory accesses. Each thread operates on one element of an idea. Specifically, the squared difference between an idea and a center at one dimension is calculated by each thread, and parallel reduction of summation is employed to sum the values of all squared differences. Due to the typically small number of idea groups, a shallow loop is employed to compute the distances from an idea to all group centers.

After labeling all the ideas with the index of its nearest group center, **kernel(B)** is used to find every group's best idea that is set as the representation of this group. In this kernel, block number is set as M and each block works on finding the best idea for one group. Specifically, i^{th} thread in j^{th} block identifies whether i^{th} idea belongs the j^{th} group based on the label of i^{th} idea which was produced by **kernel(L)** and stored in the global memory. Parallel reduction is utilised here to compare all the ideas belongs to the j^{th} group and find the idea with the best objective function value. Thus, the blockDim.x of **kernel(B)** is N and blockDim.y is 1. As to data storage, a large number of registers and the shared memory are used in this kernel for the parallel reduction operation. Then the ideas are clustered based on specific clustering algorithms.

6.4.1.3 Divergent operator

kernel(S) implements the divergent operator to generate new ideas. In this kernel, each thread handles one element of an idea. Accordingly, the block size and number are exactly the same as the **kernel($\mathbf{I}_p$)**. The shared memory is used to store some repeatedly used data, such as the centers of groups.

After generating new ideas, **kernel(E)** is employed to evaluate the objective function values of all new ideas. Then, **kernel(G)** is employed to combine the old and new ideas to produce the population for the next generation. This kernel uses one thread to compare the objective function value of an old idea with that of a newly generated idea. If the new idea is better, D' threads updates all elements of the old idea with the newly generated idea. Otherwise, D' threads do nothing. Therefore, the block size and number are the same as **kernel($\mathbf{I}_p$)**. When this kernel finishes, GPU-based MBSO checks whether to stop the algorithm. If some stopping

criterion is met, the algorithm terminates. Otherwise, stream synchronization is involved to ensure the completion of **kernel($\mathbf{I}_c$)** before starting the next generation.

6.4.2 Numerical Analysis

Here, we compare the performance of GPU-MBSO with the sequential MBSO on 30 CEC2014 benchmark problems in terms of solution quality and computing speed.

6.4.2.1 Experiments Setup

Experiments are conducted on a PC equipped with an Intel Xeon E5-2630 CPU at 2.30 Ghz and a NVIDIA Geforce GTX Titan GPU with 6GB of GDDR5 global memory. GTX Titan supports compute capability 3.5, which has 2880 SPs evenly deployed in 15 SMXs, i.e. each SMX consists of 192 SPs. Our development environment is made of Windows 7 operating system, CUDA toolkit 7.5 and Microsoft Visual Studio 2013.

The parameters are set as [120, 121] mentioned. Population sizes are set to 50, 100, 500 and 1000, respectively, for each test case. For each test problem of a specific dimension, each implementation under a specific parameter setting is executed 30 times starting from different random generator seeds while all CPU and GPU based implementations share the same initial random number generator seed for any individual run. The algorithm terminates once the maximal number of function evaluations is reached, which is set to 10^4 times the problem dimension size.

6.4.2.2 Verifying the Implementation of GPU-based MBSO

Table 6.3 shows the FEVs achieved by GPU-MBSO and its reference (denoted as CPU-MBSO). According to the guidelines, we set the original sequential MBSO as the reference and 30 CEC2014 benchmark functions at $D = 10$ as the test problems. The population size is set as $NP = 50$. The AEL is set at 10^{-8} based on the instruction of CEC2014 benchmark [88]. Every problem is tested for 30 independent instances with different random seeds. In order to achieve confident conclusion, we set a high $T = 95\%$. The significantly biased outputs indicate that

Table 6.3: Mean FEVs of GPU-MBSO and CPU-MBSO and their biases before applying correctness verification guidelines.

func	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}
CPU-MBSO	692.756	1121.3	17.274	1.624	20.363	1.535	0	15.235	14.652	441.763
GPU-MBSO	685.954	1185.2	17.534	1.862	20.352	1.723	0	16.852	14.474	440.235
bias	6.802	63.9	0.260	0.238	0.011	0.188	4.8×10^{-9}	1.617	0.178	1.528
#corr.	0/30	0/30	0/30	0/30	0/30	0/30	26/30	0/30	0/30	0/30
func	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}	f_{16}	f_{17}	f_{18}	f_{19}	f_{20}
CPU-MBSO	1786.83	0.5262	0.0732	0.4286	0.0663	0.2578	1023.094	925.673	0.4523	0.2035
GPU-MBSO	1753.62	0.4673	0.0867	0.5979	0.0760	0.3482	946.563	962.454	0.6483	0.1998
bias	33.304	0.0598	0.0135	0.1693	0.0097	0.0904	76.531	36.781	0.1960	0.0037
#corr.	0/30	0/30	0/30	0/30	0/30	0/30	0/30	0/30	0/30	0/30
func	f_{21}	f_{22}	f_{23}	f_{24}	f_{25}	f_{26}	f_{27}	f_{28}	f_{29}	f_{30}
CPU-MBSO	20.764	100.526	342.628	0	273.539	144.244	400.462	530.632	83082.4	14286.4
GPU-MBSO	20.457	100.532	342.692	0	274.622	141.452	400.843	530.668	83126.8	14085.3
bias	0.307	0.006	0.064	7.3×10^{-7}	1.083	2.792	0.381	0.036	44.4	201.1
#corr.	0/30	0/30	0/30	22/30	0/30	0/30	0/30	0/30	0/30	0/30

Table 6.4: Mean FEVs of GPU-MBSO and CPU-MBSO and their biases after applying correctness verification guidelines.

func	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}
CPU-MBSO	687.02	1170.9	17.536	1.252	20.342	1.675	0	12.659	14.423	440.675
GPU-MBSO	687.02	1170.9	17.536	1.252	20.342	1.675	0	12.659	14.423	440.675
bias	2.3×10^{-12}	6.2×10^{-9}	0	5.4×10^{-12}	0	4.6×10^{-10}	1.3×10^{-11}	0	0	1.9×10^{-7}
#corr.	30/30	30/30	30/30	30/30	30/30	30/30	30/30	30/30	30/30	26/30
func	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}	f_{16}	f_{17}	f_{18}	f_{19}	f_{20}
CPU-MBSO	1779.5	0.3562	0.0436	0.7383	0.0584	0.2385	987.438	982.90	0.5752	0.1947
GPU-MBSO	1779.5	0.3562	0.0436	0.7383	0.0584	0.2385	987.438	982.90	0.5752	0.1947
bias	6.6×10^{-7}	0	0	0	0	0	8.2×10^{-9}	4.6×10^{-7}	0	0
#corr.	26/30	30/30	30/30	30/30	30/30	30/30	29/30	28/30	30/30	30/30
func	f_{21}	f_{22}	f_{23}	f_{24}	f_{25}	f_{26}	f_{27}	f_{28}	f_{29}	f_{30}
CPU-MBSO	20.563	100.53	342.62	0	272.69	140.66	400.73	530.45	83102.7	14175.2
GPU-MBSO	20.563	100.53	342.62	0	272.69	140.66	400.73	530.45	83102.7	14175.2
bias	0	5.1×10^{-7}	5.8×10^{-12}	0	0	0	5.8×10^{-11}	8.2×10^{-9}	1.4×10^{-6}	6.0×10^{-7}
#corr.	30/30	25/30	30/30	30/30	30/30	30/30	30/30	27/30	20/30	22/30

the correctness is unclear; specifically, it only runs correctly at 48 out of 900 (5.4%) instances. Here are the actions we conducted based on the proposed guidelines:

- The original sequential CPU-MBSO is implemented based on C/C++ as the reference. The correctness of CPU-MBSO is carefully verified to insure it outputs correctly.

- The original RNG of GPU-MBSO is based on cuRAND which is significantly different with the one used by CPU-MBSO. Thus, the cuRAND-based RNG of GPU-MBSO is re-

placed with the default C/C++ one that is used by CPU-MBSO. Moreover, CPU-MBSO and GPU-MBSO generate sufficient random numbers and store them at CPU RAM in the beginning of the algorithm to insure that each random number is used exactly the same between two implementations. Then GPU-MBSO transfers these random numbers to GPU global memory. When MBSO needs random numbers, both implementations load from existing random numbers sequence instead of generating real-time random numbers.

- According to the work [121], CPU-MBSO updates the population in a asynchronous way, so the race condition will happen for GPU-MBSO. Therefore, we replace the update mechanism with a synchronous way; specifically, the population will be updated unless all the new individuals are generated and evaluated. Since there is no other operation that will cause race condition, it does not need action to avoid the race condition other than the modification of population mechanism.

- The parallel reduction of summation is employed to accumulate the values when calculating the distance from an individual to the cluster center. Therefore, in order to avoid its influence, we implement a CPU-based distance calculation operation and transfer all the individuals to CPU host memory for distance calculation. Finally, the distance values are transferred back to GPU global memory for labeling.

- The GPU-based benchmark functions are replaced with the original sequential implementations. Specifically, after GPU-based MBSO generates the new solutions, the population will be transferred from the GPU global memory to CPU RAM. Then the sequential benchmark functions are employed to evaluate their fitness values. Finally, the population that are updated with new fitness values are transferred back to the GPU global memory for the next generation.

After employing these guidelines, we conduct the experiments and list the FEVs for GPU-MBSO and its reference at Table 6.4. It can be observed that GPU-MBSO and CPU-MBSO perform very similar at most test problems. Namely, GPU-MBSO performs correctly for 863 out of 900 (95.9% $>$ 95%) instances. Therefore, the correctness of GPU-MBSO is verified

Table 6.5: The average computing time (seconds) of CPU-based MBSO (denoted as CPU-MBSO) and GPU-based MBSO (denoted as GPU-MBSO) on 30 test functions with three dimensions ($D = 10, 50$ and 100) and four population sizes ($NP = 50, 100, 500$ and 1000). Total FEs are $D * 10^4$.

	Algorithm	$NP = 50$	$NP = 100$	$NP = 500$	$NP = 1000$
$D = 10$	CPU-MBSO	0.72s	0.72s	0.73s	0.73s
	GPU-MBSO	0.38s	0.28s	0.23s	0.25s
	Speedups	1.85	2.93	7.95	14.82
$D = 50$	CPU-MBSO	18.6s6	18.58s	19.10s	19.33s
	GPU-MBSO	2.32s	1.86s	0.72s	0.61s
	Speedups	8.48	11.98	23.58	28.31
$D = 100$	CPU-MBSO	82.60s	82.54s	83.22s	83.62s
	GPU-MBSO	4.47s	3.38s	2.38s	2.18s
	Speedups	18.67	25.00	36.79	40.61

and the original GPU-MBSO can be employed confidently to accelerate the time-consuming optimisation problems.

6.4.2.3 Performance Analysis on Speedups

As the implementation of GPU-MBSO is successfully verified, it's safe to study the acceleration. Table 6.5 reports the computing time of the CPU-based sequential MBSO and the GPU-based MBSO across varying population sizes ($NP = 50, 100, 500$ and 1000) and under different problem dimensions ($D = 10, 50$ and 100), as well as the speedup of GPU-based MBSO. Major observations are as follows:

- The parallel GPU-based MBSO consistently demonstrates the superior computing speed over the sequential MBSO with respect to any population size and any problem dimension.
- Given a specific problem dimension, as the population size increases, the speedup of the parallel GPU-based MBSO over the sequential MBSO keeps increasing remarkably.

- Given a specific population size, as the problem dimension increases, the speedup ratio of the parallel GPU-based MBSO over the sequential MBSO is consistently increasing.

Here, the most existing finding is that increasing population can always improve the speedups, especially for a small dimension. Given a specific problem dimension that corresponds to a specific maximum number of function evaluations, the population size determines the total number generations (main loops) and accordingly the execution times of those kernels involved in the main loop. As the population size increases, the number of main loops decreases, which potentially reduces the total computing time. It has a great significance for EAs to apply a large population. Namely, a larger population can only improve the solution quality of CPU-based EAs, while it benefits to GPU-based EAs in terms of both computing speed and solution quality. It makes GPU-based EAs ideal for solving very complex and large-scale problems.

However, the speedups increase slightly when continuously increasing a population that is already very large, especially the problem dimension is large.

- Given a small dimension ($D = 10$), speedups increase significantly when population increases from small ($NP = 50$) to large ($NP = 1000$).

- Given a large dimension ($D = 50$ or 100), speedups increase significantly when population increases from small ($NP = 50$) to medium ($NP = 500$) and increase slightly when population increases from medium ($NP = 500$) to large ($NP = 1000$). It is because problems with a large population easily occupy the GPU device and further increasing the population size can not improve the parallelism.

Summarily, when solving very complex or large-scale problems using a large population, a part of population have to wait for the vacant of GPU threads, resulting in difficulties in the scalability. In order to address this issue, we will discuss how to improve the scalability of GPU-based EAs by utilising more GPU devices in Chapter 7.

6.5 Conclusions

This chapter designs a set of guidelines to verify the correctness of GPU-based EAs. Specifically, an example of migrating the PSO from CPU based coding to the GPU environment is firstly given as an example to show the importance of correctness verification. In addition, some GPU-inherent issues, which influence the output of GPU-based EAs including the library functions, the numerical precision, and the race condition, are examined one by one. To cope with the issues mentioned above, a set of guidance is proposed to verify the correctness of the GPU-based EAs. Finally, a working example based on GPU-MBSO is presented to examine the effectiveness of the proposed guidelines. From the working example, we also present that EAs based on a single GPU will suffer a weak scalability due to the limited computing power.

Chapter 7

SPEO based on Multiple GPUs at GPU-enabled HPC ($\text{SPEO}_{\text{HPC}_{\text{gpu}}}$)

7.1 Introduction

In Chapter 6, we also shown that a single GPU can only offer limited computing power which fails to speedup GPU-based EAs with increasingly large population size or problem dimension. As a result, on-demand GPUs on GPU-enabled HPC, which can provide scalable GPU computing power, becomes ideal for solving extremely large-scale problems by using a large population size.

However, utilizing on-demand GPUs on GPU-enabled HPC is associated with a new challenge apart from the ones that are faced in $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$. Specifically, CPU and GPU work cooperatively in heterogeneous architecture, and thus tasks can be assigned to either GPU or CPU or a combination of both. Therefore, finding the optimal mapping choice is essential to achieve scalability when running parallel EAs on GPU-enabled HPC. So far, existing works fail to carry out this essential scalability because they are only designed for a single GPU [79–82] or a small fixed number of GPUs [83–87].

In this chapter, we propose the SPEO based on GPU-enabled HPC ($\text{SPEO}_{\text{HPC}_{\text{gpu}}}$). As extended is from $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$, the $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ inherits the crucial asynchronous migration of $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$, and additionally introduces a dual control mode to further improves the scal-

ability. The proposed framework is implemented with a GPU-based implementation of DE algorithm and its performance is evaluated on eight composition functions of CEC2014 benchmark at NCI GPU-enabled HPC. Experimental results demonstrate that it achieves linear speedups on 2 to 64 GPUs which indicates the excellent scalability. Results also show that the $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ outperforms the $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ and a state-of-the-art CPU-based parallel EA with the same computational budget of USD 1, 10, 50 and 100.

The remainder of this paper is organised as follows. Section 7.2 describes the proposed $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ framework. Experimental results over numerical optimisation problems are presented in Section 7.3. Section 7.4 concludes this chapter.

7.2 The Proposed Method

Generally, $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ framework is also based on the island model with buffer-based asynchronous migration scheme. Thus, it has the same algorithm flow with $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$. However, from the perspective of the deployment, they are significantly different in two following aspects:

- $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ performs genetic operations on GPUs, while $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ performs them on CPU cores.
- $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ assigns an independent CPU core to perform the asynchronous migration for each island, while $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ executes the genetic operations and the asynchronous migration on the same CPU cores.

In this section, we firstly introduce how to deploy parallel EAs on GPU-enabled HPC to efficiently utilise many powerful GPU devices. Then we implement the proposed framework with a GPU-based DE and a dynamic regrouping strategy.

7.2.1 Framework

GPU computing is known as "heterogeneous" or "hybrid" computing, thus computing tasks can be assigned to either GPU or CPU or a combinations of both. Therefore, the mapping choice has a great impact on the computational efficiency of the framework taking into account that

GPU and CPU work in significantly different manners. In particular, the proposed $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ framework is designed here to realize the following four forms:

- In order to efficiently utilise increasing number of GPUs in a scalable way, the proposed framework deploys the global population over multiple GPUs based on the island model which has been shown very scalable in Chapter 4.
- As GPU is designed to undertake the compute-intensive tasks, each GPU is designed to perform parallelisable EA-related operations including crossover, mutation, evaluation and replacement.
- As GPU acts as a co-processor in GPU computing, a CPU core is necessarily assigned for each GPU to undertake GPU-related operations including launching kernel functions of GPU-based EAs, and synchronising data between GPU global memory and system RAM.
- When multiple GPUs are demanded, CPU cores are also responsible for communication in most cases (direct communication between GPUs is available in limited conditions, e.g. NVLINK). The proposed framework applies a dual control mode which utilises an extra CPU core for each GPU to perform communication tasks.

7.2.1.1 $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ and the infrastructure of GPU-enabled HPC

The population distribution model of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ is presented at the inside circle of Figure 7.1. Here, an island with N_s individuals are deployed to each of M_{gpu} GPUs; thus the global population contains totally $NP = N_s \cdot M_{gpu}$ individuals. These islands exchange information via a specific migration topology represented by solid lines in Figure 7.1. It also can be observed that all pairwise islands are connected by either the dotted and solid lines, which indicates that the proposed framework is not limited by a specific topology and supports even the most complex topology (fully connected).

The deployment of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ on a common infrastructure of GPU-enabled HPC is presented at the outside ring with the shadow in Figure 7.1. Enclosed by the dotted ellipse, two CPU cores, a GPU, RAM and communication system compose a single computing unit

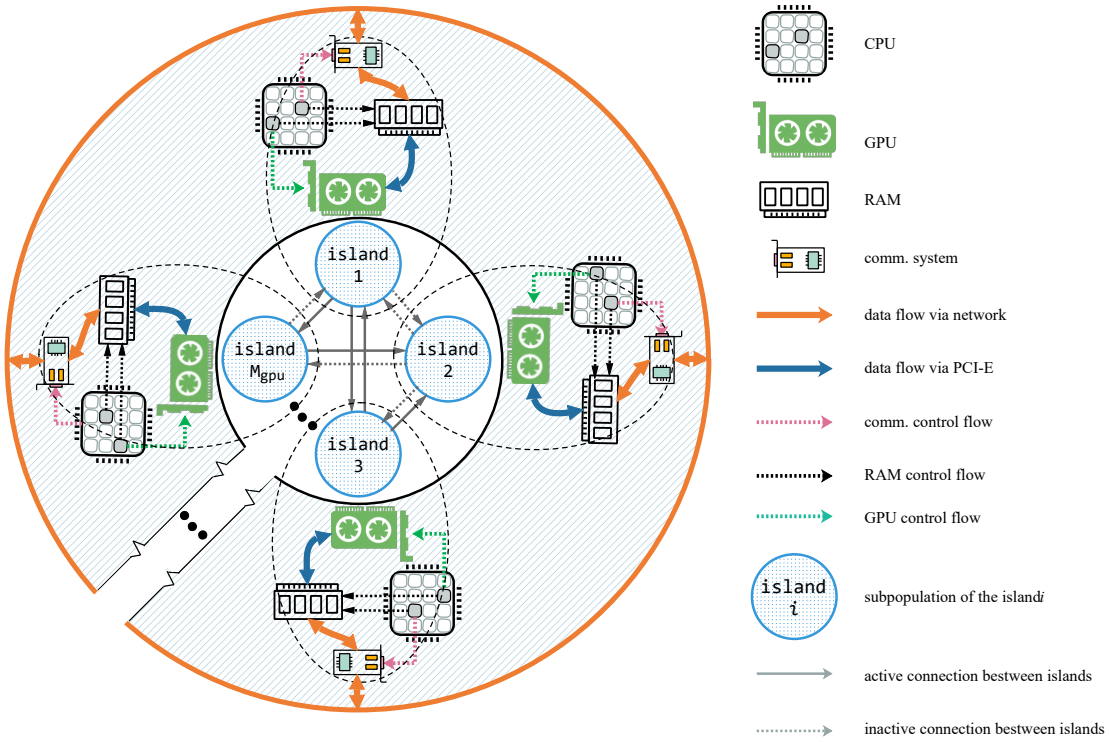


Figure 7.1: The deployment of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ on the infrastructure of GPU-enabled HPC.

that undertake all operations to evolve a single island. Without loss of generality, the communication system can be a mixture of network (e.g. InfiniBand), point-to-point processor interconnect (e.g. QPI) and some specific devices applied by GPU-enabled HPC. Here, two CPU cores are utilised by the dual control mode to separate the GPU control tasks with the management of communication tasks. Specifically, one CPU core undertakes all GPU-related tasks (green dotted arrow) and manages data flow between GPU global memory and system RAM (blue solid arrow); while another CPU core controls the communication system (pink dotted arrow) and manages the data flow between the system RAM of all computing units via the communication system (orange solid line and arrow).

The $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ framework also employs the buffer-based asynchronous migration strategy that is proposed for the $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$. Specifically, the migration strategy performs following three tasks: 1) It updates the island when the current island seeks new information to better

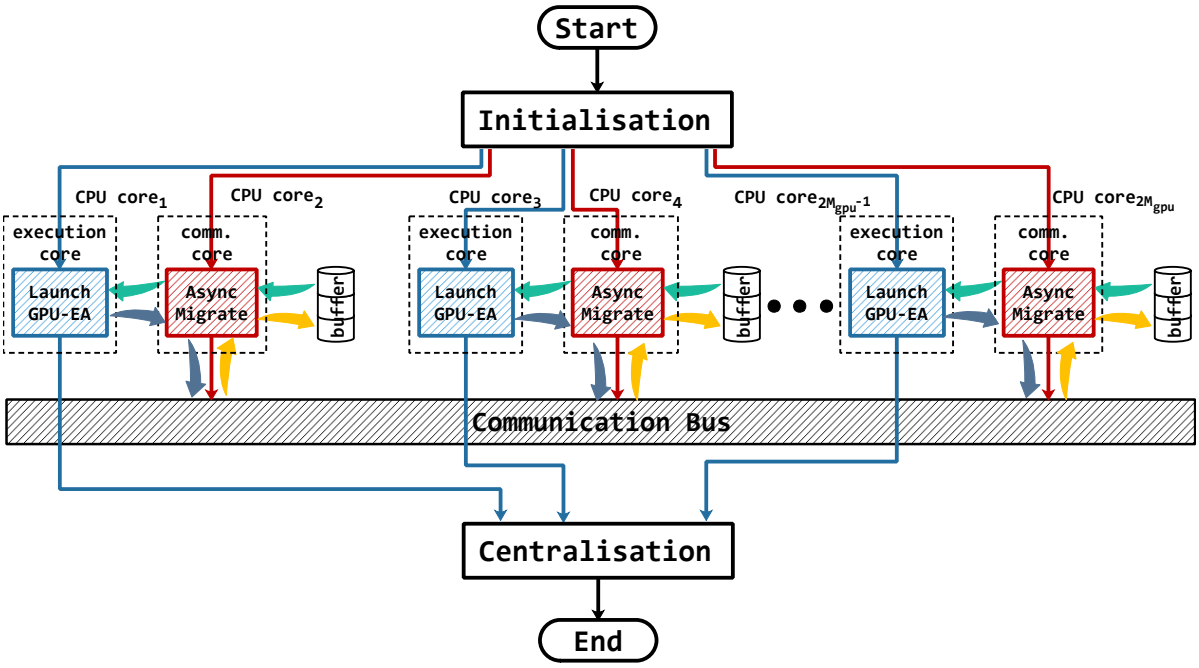


Figure 7.2: Framework of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ with the dual control mode.

explore in the searching space (green arrow). 2) It receives newly arrived immigrants that are sent from other islands (yellow arrow). 3) It selects emigrants from the current island and sends them to other islands via the communication bus (grey arrow). In order to improve the computing efficiency and avoid the communication congestion when an increasing number of GPU devices are utilised, a dual control mode is proposed for $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$.

7.2.1.2 Dual control mode

In this chapter, we propose a dual control mode which improves the computational efficiency when increasing number of GPUs are used. The proposed dual control mode utilises two CPU cores for each GPU to manage communication tasks and to control GPU separately. Figure 7.2 presents the diagram of the dual control mode. It can be observed that M_{gpu} CPU cores (core 1, 3, 5 ..., $2M_{gpu} - 1$) are initialised as execution cores to control the GPU to perform GPU-based EA operations on islands (the blue line and box) and another M_{gpu} CPU cores (core 2, 4, 6 ..., $2M_{gpu}$) are initialised as communication cores to manage the communication

for migration between islands (the red line and box). A buffer is also designed to store the imported immigrants from other islands. When all execution cores finish the evolving of islands, a centralisation operation is performed to collect all islands from M_{gpu} GPUs and the best solution is finally outputted.

The execution core acts very similar with existing parallel EAs based on a single GPU except following two aspects. 1) It imports some immigrants from the buffer and insert them into the island (green arrow). 2) It selects some emigrants from the current island and exports them to the communication core for further sending to other islands (gray arrow).

The communication core manages massive communication tasks that share information with other islands via the communication system in the following two aspects. 1) It receives immigrants from communication cores belonging to other islands. Then the buffer is update with these immigrants by using some specific strategies such as the diversity preserving buffer that is proposed in Chapter 3 (yellow arrow). 2) It sends the emigrants, which is exported from the execution core, to the communication cores belonging to other islands via the communication system (gray arrow).

In this way, each communication core links with all other communication cores but links only one execution core which controls a single GPU. When the number of GPUs increases, the execution core only concentrates on controlling its corresponding GPU regardless of the increasing workload of communication. Thus, the proposed framework can always work efficiently even though increasing GPUs are demanded.

It will not bring any extra cost because most modern GPU-enabled HPCs package a GPU with several CPU cores and only charge for the utilisation of GPU. For example, each K80 GPU is packaged with 3 CPU cores at NCI[1] and 4 CPU cores at AWS EC2 P2 instance[2], respectively.

7.2.2 Implementation of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$

As each computing unit concludes two CPU cores and one GPU, the proposed $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ have to be implemented correspondingly based on specific devices and functions.

[1] https://opus.nci.org.au/display/Help/GPU+User+Guide
[2] https://aws.amazon.com/ec2/pricing/on-demand/

7.2.2.1 Implementation of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ on the execution core

Algorithm 4: implementation $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ of i^{th} execution core, $i \in [1, M_{gpu}]$

input : $\{N_s, I, R_m\}$

Perform **kernel(I)** and **kernel(E)** to initialise the island $\mathbf{P}_{i,G}^{gpu}$ on GPU global memory;
$G = 0$, group size $S \leftarrow N_s$;
while *the predefined termination criteria is not met* **do**
  Perform **kernel(MC)** to generate trial vectors $\mathbf{T}_{i,G}^{gpu}$ based on $\mathbf{P}_{i,G}^{gpu}$, the group size S and DE parameters;
  Perform **kernel(E)** to evaluate fitness values of $\mathbf{T}_{i,G}^{gpu}$;
  Perform **kernel(R)** to generate $\mathbf{P}_{i,G+1}^{gpu}$ based on $\mathbf{P}_{i,G}^{gpu}$ and $\mathbf{T}_{i,G}^{gpu}$;
  $G \leftarrow G + 1$;
  `/* Export emigrants to communication core */`
  if $mod(g, I) = 0$ **then**
    copy $\mathbf{P}_{i,G}^{gpu}$ at GPU global memory to $\mathbf{P}_{i,G}^{cpu}$ at CPU RAM;
    EM $\leftarrow$ select $R_m * N_s$ emigrations from $\mathbf{P}_{i,G}^{cpu}$;
    Non-blocking send **EM** to i^{th} communicator by $mpi\_Isend()$;
  end
  if *immigrations arrive* ($mpi\_Iprobe() = 1$) **then**
    `/* Update island */`
    IM $\leftarrow$ Receive immigrations from i^{th} communicator using $mpi\_Recv()$;
    copy $\mathbf{P}_{i,G}^{gpu}$ at GPU global memory to $\mathbf{P}_{i,G}^{cpu}$ at CPU RAM;
    Merge $\mathbf{P}_{i,G}^{cpu}$ and **IM**;
    `/* Perform dynamic regrouping */`
    Randomly generate the group size S;
    Randomly shuffle $\mathbf{P}_{i,G}^{cpu}$;
    copy $\mathbf{P}_{i,G}^{cpu}$ at CPU RAM to $\mathbf{P}_{i,G}^{gpu}$ at GPU global memory;
  end
end

The execution core is responsible to launch GPU kernel functions and to perform some light-compute tasks. Algorithm 4 presents the implementation of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ on the execution core. Apart from GPU kernels that will be discussed later, three assistant operations are implemented at as follows:

Export emigrants to the communication core: It firstly copies the island at the GPU

global memory to CPU RAM. Then it selects the best $N_s * R_m$ emigrants from the island and passes them to its communication core which will further sharing this information with other islands. Here, the passing of emigrants from the execution core to the communication core is also via the communication system, while users can choose some other ways such as via the shared memory. Specifically, a non-blocking sending MPI API (MPI_Isend()) is employed (the line 11 at Algorithm 4). Thus, the execution core will immediately conduct the further instructions, instead of wait for the successful sending. It can save computing budget and avoid the waiting of execution core if the communication core is busy now and cannot receive information immediately.

Update island: It checks the communication system and will immediately receive the arrived immigrants that are sent from the buffer of the communication core. Then it copies the island at GPU global memory to CPU RAM and merges the islands with the immigrants.

Perform dynamic regrouping: When implementing parallel EAs based on GPUs, the data volume (a.k.a. the population size) impacts the searching behavior as well as the computational efficiency of GPU. Specifically, many works [79, 174, 240] have indicated that a large population size significantly improves the computational efficiency due to the better utilisation of thousands of GPU cores. However, a large population may suffer the difficulties in convergence. Thus the dynamic regrouping strategy is introduced and implemented to insure the exploitation capability of a large island and to maintain the computational efficiency on GPU meanwhile. The dynamic regrouping strategy are described as follows:

- When the dynamic regrouping strategy is performed, a group size S is randomly generated and the large island is divided into several small groups based on this group size. The group size is generated as $S = 2^n * S_{min}$, where interger n is randomly selected from $[0, log_2(\frac{N_s}{S_{min}})]$ and S_{min} is the minimal size pre-defined by users based on the specific EAs.

- A group is the minimal evolution unit so that genetic operations such as crossover only occurs between individuals belonging to the same group. As a result, groups do not exchange any information once a certain grouping result is obtained. To share information between different groups belonging to the same island, dynamic regrouping strategy is

performed frequently during the life circle of evolution to produce different grouping results in terms of group sizes and group members.

- When implementing the dynamic regrouping, the individuals of an island are assigned into groups in turns. For example, if 16 individuals are grouped into 4 groups, the individual 1 to 4 are group 1, the individual 5 to 8 are group 2 and so on. Thus, to generate different grouping results in terms of group members, the individuals of an island are randomly shuffled (see the line 18 at Algorithm 4) when the dynamic regrouping is performed.

After the dynamic regrouping is performed, the shuffled island at CPU RAM is copied to GPU global memory and it steps into the next generation.

7.2.2.2 Implementation of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ on the communication core

Algorithm 5 illustrates the implementation on the communication core. The communication core starts with initializing the diversity preserving buffer $\mathbf{B}_i$ and follows a loop and will not stop unless it meets termination condition.

Communicate with the execution core: The i^{th} communication core stores emigrants **EM** sent from the i^{th} execution core. Then it selects the first $R_m * N_s$ immigrations from the buffer $\mathbf{B}_i$ and sends them back to the i^{th} execution core immediately. After that, it selects several recipients $\mathbf{r}$ based on the improved dynamic topology the same as $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$.

Communicate with other communication cores: The communication core alao receives emigrations from other communication cores and inserts them into the diversity preserving buffer $\mathbf{B}_i$ in the same way as $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$. After that, **EM** that are the latest emigrants received from the execution core are sent to other communication core of $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$. Specifically, it checks the availability of communication system and the emigrants **EM** are sent to the first recipient island r in $\mathbf{r}$. After that, the recipient r is deleted from $\mathbf{r}$. Otherwise, it finishes current iteration and starts the next.

The implementations of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ on execution core and communication core are programmed with C/C++ and the communication operations are implemented using the MPI message-passing programming model [229]. The non-blocking point-to-point communication

Algorithm 5: i^{th} communication core, $i \in [1, M_{gpu}]$

input : { C_b, R_c, R_m, M_{gpu}}

$\mathbf{B}_i \leftarrow \emptyset$;
while *Receive flag of finish* **do**
    /* Communicate with the execution core */
    if *emigrations arrive from the* i^{th} *execution core (*$mpi\_Iprobe() = 1$ *and it is sent from execution core)* **then**
        EM ← Receive emigrations;
        IM ← Select the first $R_m * N_s$ immigrations in $\mathbf{B}_i$;
        Non-blocking send **IM** to i^{th} execution core using $mpi\_Isend()$;
        r ← randomly select $R_c * (M_{gpu} - 1)$ recipient islands;
    end
    /* Communicate with other communication cores */
    if *immigrations arrive from other communication cores (*$mpi\_Iprobe() = 1$ *and and it is sent from other communication cores)* **then**
        IM ← Receive immigrations from communicators using $mpi\_Rend()$;
        $\mathbf{B}_i$ ← Update the buffer by diversity preserving described at Algorithm 1 in Chapter 4;
    end
    while *Communication system is available and* **r** *is not empty* **do**
        Send **EM** to the first island in **r** using $mpi\_Isend()$;
        Delete the first island from **r**;
    end
end

functions MPI_Isend() and MPI_Iprobe() are used to send emigrants and check incoming immigrants, respectively. The blocking MPI_Recv() is employed to receive immigrants once MPI_Iprobe() indicates any immigrant arrives.

7.2.2.3 Implementation of parallel EA on GPU

The $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ accepts most GPU-based parallel EAs as the specific implementation to evolve each island. Here, a GPU-based implementation of DE is employed and the detailed implementation of GPU-based DE is presented at Figure 7.3 and is introduced as follows:

kernel(I) initializes the population in the searching space and writes it into the global memory. This kernel utilizes uniform random numbers generated by the cuRAND library.

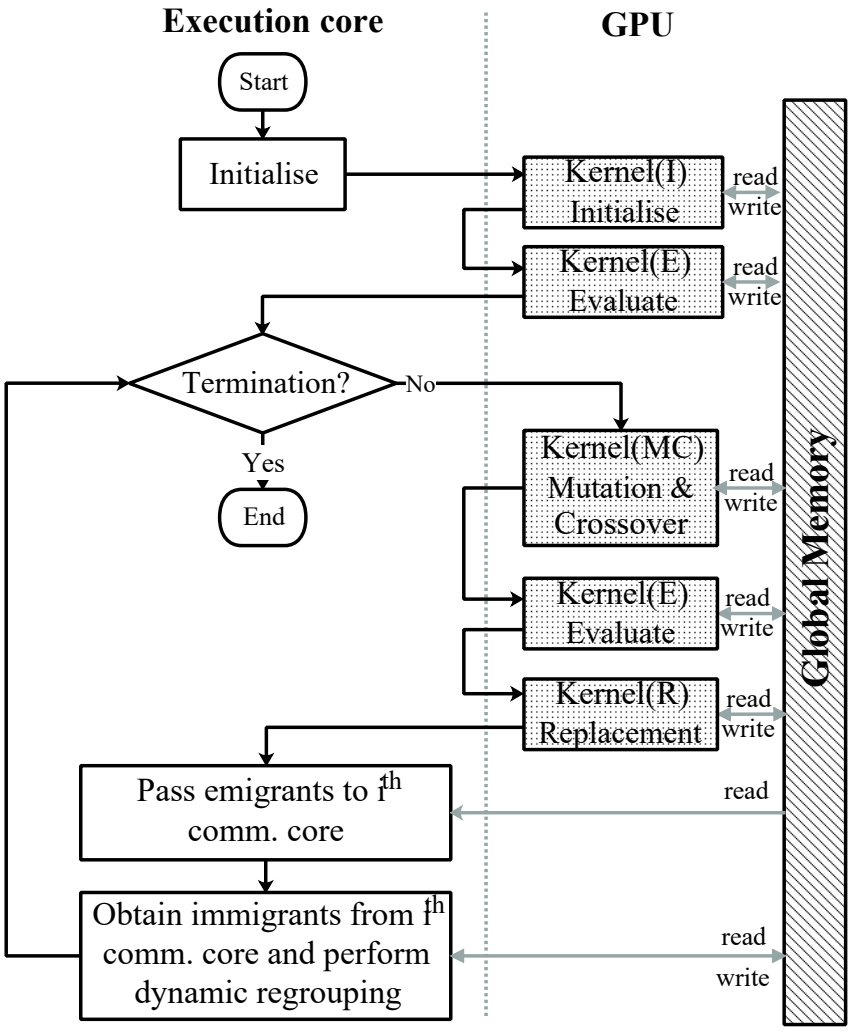


Figure 7.3: Implementation of parallel DE on GPU. I represents the interval for active migration and G is the current generation.

kernel(E) is employed to evaluate the quality of population. This kernel calculates the objective function values of the individuals and writes them into the global memory. The objective function is defined according to the problem being solved and thus has varying complexity. Since the benchmark functions contain the addition and/or multiplication operations on all elements of a population member, we employed the parallel reduction to make the sum and production calculations. It is worth of noting that the objective function evaluation could

be the most time-consuming, especially when the problem and population size grow. Therefore, the effective parallelization of this part on GPU may lead to remarkable computational speedup. In addition to the global memory, **kernel(E)** also uses the shared memory to store the data that needs to be frequently used.

kernel(MC) perform crossover and mutation of DE algorithm to generates trial vectors for each individual using its respective DE strategy, CR and F values. As a stochastic searching algorithm, GPU-based DE employs cuRAND library as the RNG to generate trial vectors. The generated trial vectors are stored at the global memory of GPU. In order to generate trial vectors based on the dynamic regrouping strategy, **kernel(MC)** selects random and differential vectors based on the group size S. Specifically, for any target vector, its random and differential vectors are always located in the same group with the target vector.

kernel(R) compares each member in the current population with its corresponding trial vector generated by **kernel(MC)** in terms of their objective function values, and writes the fitter one and its corresponding objective function value into the global memory.

The block size and number of all the kernels are configured exactly the same. Specifically, the blockDim.x of all kernels is $D(dim)$, where $D(dim) = 2^p, 2^{p-1} < dim < 2^p$. The blockDim.y is $max(1, \lfloor \frac{maxTread}{D(dim)} \rfloor)$, where $maxThread$ is the maximum number of resident threads per block which is typically of 1024. The number of block is $\lceil NP * \lfloor \frac{D(dim)}{maxTread} \rfloor \rceil$.

Here, all genetic operations including initialisation, crossover, mutation, evaluation and replacement are implemented with CUDA-C. The correctness of these kernel function is carefully verified based on the guidelines proposed in Chapter 6. All kernel functions are launched by the execution core that can be observed at Algorithm 4.

7.3 Experiments

In this section, the performance of the proposed $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ is examined in the following aspects:

- Whether the proposed $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ is scalable with increasing GPU devices in terms of the computational speed and the solution quality.

Table 7.1: Configurations of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$.

Parameter	Notation	Value
Island size	N_s	64, 128 ..., 2048, 4096
Number of GPUs	M_{gpu}	2, 4, 8, 16, 32, 64
Migration interval	I	100
Connection rate	R_c	25%
Migration rate	R_m	5%
Buffer capacity	C_b	N_s
Minimal group size	S_{min}	4
Evolutionary algorithm	DE	rand / 1 / bin, CR / F=0.9 / 0.5

- Whether the dual control mode improves the computational efficiency of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$.
- Whether the dynamic regrouping strategy improves the solution quality when a large island size is employed.
- Whether $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ outperforms the $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ and the state-of-the-art CloudDE with the same computational budgets.

7.3.1 Test Problems

To examine the performance of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$, 8 difficult test functions (complex composition function $f_{23} - f_{30}$ from CEC2014 [88] benchmarks) which are briefly introduced at Chapter 3.2.1 are selected as our test problems. In this chapter, we only test the largest dimension $D = 100$ to demonstrate the potential of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ on solving complex and difficult problems. Each dimension is bounded within [-100, 100]. 8 test problems are implemented with CUDA-C to efficiently execute on GPUs.

7.3.2 Experimental settings

7.3.2.1 Configurations

Table 7.1 presents the configurations of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$. In order to avoid losing generality, DE is configured with standard settings: rand/1/bin, $CR = 0.9$ and $F = 0.5$ according to the work [47]. Since DE/rand/1/bin requires at least 3 individuals to reproduce offspring, the minimal group size is set as $S_{min} = 4$ to guarantee the island size divisible by S_{min}. The island size is configured from small (64) to large (4096) to investigate its influence on performance. Other parameters are set based on the work of $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$.

Since $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ will be performed with a very large population, a normally maximal FEs is no longer sufficient for convergence. Moreover, achieving an excessively FEs in a reasonable time is now available for $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ because it runs far faster than traditional sequential or parallel EAs based on CPU. Therefore, the total FEs are set as $D * 10^7 = 10^9$ in this work.

7.3.2.2 Computing platform

All experiments are conducted on NCI GPU node. Each NCI GPU node is comprised by 4 Nvidia Tesla K80 GPUs and each K80 is comprised by 2 GPUs (2496 CUDA cores per GPU). Despite that each GPU node has 8 GPU cards. it has two Intel CPUs that are Haswell E5-2670v3 with 12 CPU cores or Broadwell E5-2690v4 CPUs with 14 CPU cores. Since NCI forces users to allocating 3 CPU cores for for each GPU, the dual control mode only uses two of them for each GPU, and leave the rest one core vacant. In this work, up to 64 GPU are utilised to comprehensively investigate the performance of our $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$.

7.3.2.3 Evaluations

The same as evaluation criterion of $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$, the solution quality is measured by the FEVs [88]. The computational efficiency is measured by the speedups of the parallel implementation of the algorithm executed on the GPU-enabled HPC computing platform against its sequential counterpart executed on the single CPU core. The definition of speedup is given as

$$speedup = \frac{T_s}{T_{ngpu}}$$

Table 7.2: Average computational time (hh:mm:ss) of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ on 8 test problems with increasing GPUs ($M_{gpu} = 2, 4, 8, 16, 32$ and 64) and various island sizes ($N_s = 64, 128, 256, 512, 1024, 2048$ and 4096). Total FEs are $D * 10^7 = 10^9$.

N_s	2 GPUs	4 GPUs	8 GPUs	16 GPUs	32 GPUs	64 GPUs
64	01:33:36	00:46:06	00:23:04	00:11:29	00:05:46	00:02:53
128	00:47:49	00:23:28	00:11:42	00:05:55	00:02:55	00:01:29
256	00:25:21	00:12:23	00:06:11	00:03:10	00:01:33	00:00:47
512	00:16:31	00:08:05	00:04:06	00:02:01	00:00:58	00:00:30
1024	00:11:39	00:06:04	00:03:00	00:01:26	00:00:43	00:00:22
2048	00:09:07	00:04:55	00:02:24	00:01:09	00:00:35	00:00:18
4096	00:08:14	00:04:14	00:02:08	00:01:03	00:00:30	00:00:15

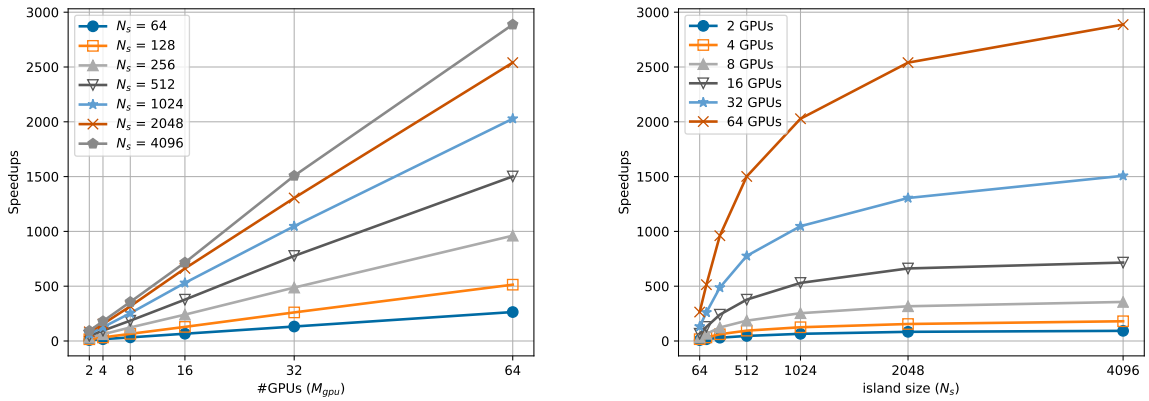
where T_s and T_{ngpu} are the average execution time of the sequential and the GPU-based implementations over n GPUs on all test problems, respectively.

7.3.3 Scalability Analysis

Weak scaling test [230, 231] in parallel computing allows us to look at the capability of an algorithm to solve larger or more complicated problems in conjunction with the use of more resources. Based on this definition, Liu [69] extended the weak scaling test to demonstrate how the execution time varies with the increasing number of devices when the population size per device is fixed. Therefore, we use an increasing number of GPUs and fix the island size to examine the improvements of the proposed framework in terms of speedup and solution quality. Since the island size has a great impact on the performance of GPU-based EAs, the weak scaling test is repeatedly conducted using various island sizes ranging from small (64) to large (4096).

7.3.3.1 Computational time

Table 7.2 presents the average computational time of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ on 8 test problems with an increasing number of GPUs ranging from 2 to 64, and various island sizes ranging from small ($N_s = 64$) to large ($N_s = 4096$). Figure 7.4a and 7.4b present the speedups of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$



(a) Speedups on increasing GPUs versus its sequential counterpart on a single CPU core

(b) Speedups with increasing island sizes versus its sequential counterpart on a single core

Figure 7.4: Scalability test of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ with different island sizes on increasing GPUs.

compared to its sequential counterpart. Based on these results, three major observations are as follows:

- According to Table 7.2, the computing time highly depends on the number of GPUs and the island size. Specifically, if $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ is executed on 2 GPUs with a small island size $N_s = 64$, it requires more than 1 hours and 33 minutes. However, it can be reduced to less than 9 minutes ($N_s = 4096$, 2 GPUs) or less than 3 minutes ($N_s = 64$, 64 GPUs).

- According to Figure 7.4a, $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ can achieve significant speedups compared to its sequential counterpart which requires about 12.7 hours on a single CPU core. For example, $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ with a 4096 island size achieves approximate 3,000x speedup on 64 GPUs.

- For a given island size, demanding more GPUs brings an approximately linear speedup which indicates the good scalability of the proposed $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ with increasing computing resources.

For a given amount of the total FEs, the number of main loops decreases when a larger global population size ($N_s * M_{gpu}$) is employed. As a result, demanding more GPUs manyfold increases

Table 7.3: Comparisons of mean FEVs with different island sizes on 64 GPUs. Significantly better value is typed in bold.

$func$	$N_s = 64$	$N_s = 128$	$N_s = 256$	$N_s = 512$	$N_s = 1024$	$N_s = 2048$	$N_s = 4096$
f_{23}	**348.235**	**348.235**	**348.235**	**348.235**	**348.235**	**348.235**	**348.235**
f_{24}	393.692	382.746	377.446	373.711	370.998	367.527	**365.487**
f_{25}	260.997	230.939	**200.0**	**200.0**	**200.0**	**200.0**	**200.0**
f_{26}	123.906	103.275	**103.237**	**103.223**	**103.226**	**103.226**	**103.232**
f_{27}	1462.154	1052.882	761.607	548.478	429.253	361.635	**343.048**
f_{28}	2473.947	**2164.827**	2230.905	2212.475	2191.477	**2172.795**	**2160.542**
f_{29}	987.252	902.647	758.817	742.37	734.014	**728.071**	779.0
f_{30}	6302.503	5231.051	4881.949	4237.765	3830.2	**3315.001**	**3280.476**

the data volume (a.k.a. the global population size) processed in parallel.

On the other hand, Figure 7.4a and 7.4b both indicate that increasing the island size can improve the efficiency. It is because a larger island size can better utilise thousands of CUDA cores belonging to a single GPU. However, increasing the island size can not always bring significant improvement of the computing speed. For example, if the island size increases from 64 to 1024, speedups on 64 GPUs increase from 10 to 2,000; if the island size continuously increases to 4096, the speedup reaches 3,000 and the upward trend flattens out. It is because thousands of GPU cores are fully occupied and further increasing the volume of data results in the queue of data processing which prevents the further improvement of computing speed.

7.3.3.2 Solution quality

Before investigating the solution quality of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ with an increasing number of GPUs, we study the impacts of different island sizes on the solution quality at Table 7.3. It presents FEVs of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ on 64 GPUs with island sizes ranging from small ($N_s = 64$) to large ($N_s = 4096$). The bold values in the table are statistically better than the normal values while all bold values are statistically similar with each other. According to Table 7.3, it can be observed that a large island size significantly improves the solution quality compared to the smaller island sizes. Therefore, $N_s = 4096$ is an ideal island size and is set as the default in the following experiments because it can achieve the best computing speed as well as the best solution quality.

Table 7.4: Comparisons of mean FEVs on different GPUs (2 GPUs to 64 GPUs) with a fixed island size $N_s = 4096$. $M_{gpu} = 2$ and $N_s = 64$ is also shown to represent DE with a normal population size. Significantly better value is typed in bold.

func	2 GPUs[1]	2 GPUs	4 GPUs	8 GPUs	16 GPUs	32 GPUs	64 GPUs
f_{23}	**348.235**	**348.235**	**348.235**	**348.235**	**348.235**	**348.235**	**348.235**
f_{24}	391.251	373.979	370.465	369.63	367.54	**366.085**	**365.487**
f_{25}	267.856	**208.433**	**200.0**	**200.0**	**200.0**	**200.0**	**200.0**
f_{26}	190.294	**103.178**	103.215	103.205	103.223	103.214	103.232
f_{27}	1995.648	506.214	404.393	365.149	362.978	**345.038**	**343.048**
f_{28}	3025.781	2328.381	2213.391	**2173.701**	2177.396	**2153.362**	**2160.542**
f_{29}	1225.65	729.282	723.627	722.521	**720.991**	724.798	779.0
f_{30}	7863.298	4248.766	4014.23	**3594.463**	**3481.64**	**3246.239**	**3280.476**

[1] island size $N_s = 64$.

The results of scalability test of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ is shown at Table 7.4 which compares the FEVs of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ on up to 64 GPUs. Results indicates that demanding more GPUs significantly improves the solution quality. The reason could be the diverse searching behavior on different GPUs which results in a better diversity of the global population. Moreover, we also show the solution quality of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ with a normal population size ($N_s = 64$ over 2 GPUs, total population size $NP = 128$). The results demonstrate that $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ with a large population can achieve significantly better solutions when compared to a normal population size.

In conclusion, the proposed $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ has an excellent scalability considering the fact that demanding more GPUs can not only remarkably improve the computing speed but also find better solutions.

7.3.4 Performance Analysis on Dual Control Mode

The dual control mode demands one more CPU core for each GPU to separate the communication and GPU launch tasks. In order to show that the extra core indeed improves the efficiency of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$, we compare computing time of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ to its variant with the single control mode (denoted as Single-$\text{SPEO}_{\text{HPC}_{\text{gpu}}}$). It assigns only a single CPU core for each GPU to process GPU kernel launch tasks and manage communication tasks. Table 7.5 presents the average computing time (T), communication cost ($comm.\%$), the relative speedup ($\frac{T_{single}}{T_{dual}}$) and

Table 7.5: Comparison of computing time (T) and communication cost (comm.%) between $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ and its variant with the single control mode (denoted as Single-$\text{SPEO}_{\text{HPC}_{\text{gpu}}}$). Aggregative statistical tests (+/≈/-) indicate $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ performs statistically better, similar and worse than Single-$\text{SPEO}_{\text{HPC}_{\text{gpu}}}$.

#GPUs	$\text{SPEO}_{\text{HPC}_{\text{gpu}}}$		Single-$\text{SPEO}_{\text{HPC}_{\text{gpu}}}$		$\frac{T_{single}}{T_{dual}}$	$+/\approx/-$
	T_{dual}	$comm.\%$	T_{single}	$comm.\%$		
2	00:08:14	6.00%	00:22:00	56.2%	2.67	1 / 6 / 1
4	00:04:14	7.02%	00:13:37	61.5%	3.45	0 / 8 / 0
8	00:02:08	7.14%	00:07:31	64.9%	3.83	1 / 6 / 1
16	00:01:03	7.43%	00:04:12	66.0%	4.31	1 / 7 / 1
32	00:00:30	7.97%	00:02:16	67.2%	4.86	3 / 4 / 1
64	00:00:15	8.45%	00:01:15	68.8%	5.33	0 / 8 / 0

the aggregative statistical results of FEVs by $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ and Single-$\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ on up to 64 GPUs. The communication cost is calculated based on the percentage of total computing time that is spent on the communication tasks by the execution core of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ and the single core of Single-$\text{SPEO}_{\text{HPC}_{\text{gpu}}}$, respectively. Three main observations are follows:

- Single-$\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ increases the communication cost from 56.2% on 2 GPUs to 68.8% on 64 GPUs. It can be inferred that Single-$\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ utilises the GPUs inefficiently because it wastes the most of computing time on communication and can not launch GPU in time when the GPU is vacant for new computing tasks. On the contrary, the communication cost of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ is steady at around 6.0%-9.0%. It indicates that the execution core will not spend more time on communication tasks even more GPUs are utilised; instead, it is always ready to launch GPU kernel functions once the GPU can undertake new computing tasks.

- $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ achieves much larger computational efficiency compared to Single-$\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ because $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ achieves up to 5.33 relative speedups. It is worth to employ dual control mode with the extra CPU cores because demanding an extra CPU core for each GPU is much easier and cheaper than demanding more GPUs on commercial computing platforms, let alone the fact that most commercial platforms provide several free CPU

Table 7.6: Comparisons of mean FEVs of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ with its variant without dynamic regrouping (denoted as Static-$\text{SPEO}_{\text{HPC}_{\text{gpu}}}$). They are executed on 64 GPUs with three island sizes ($N_s = 1024, 2048$ and 4096). Statistical tests (+/≈/-) indicate Static-$\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ performs significantly better (+), similarly (≈), or worse (-) than $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$.

$func$	$N_s = 1024$		$N_s = 2048$		$N_s = 4096$	
	$\text{SPEO}_{\text{HPC}_{\text{gpu}}}$	Static-$\text{SPEO}_{\text{HPC}_{\text{gpu}}}$	$\text{SPEO}_{\text{HPC}_{\text{gpu}}}$	Static-$\text{SPEO}_{\text{HPC}_{\text{gpu}}}$	$\text{SPEO}_{\text{HPC}_{\text{gpu}}}$	Static-$\text{SPEO}_{\text{HPC}_{\text{gpu}}}$
f_{23}	348.235	348.235 (≈)	348.235	348.326 (−)	348.235	354.931 (−)
f_{24}	370.998	365.725 (+)	367.527	398.084 (−)	365.487	466.652 (−)
f_{25}	200.0	302.069 (−)	200.0	362.304 (−)	200.0	399.229 (−)
f_{26}	103.226	103.56 (−)	103.226	103.635 (−)	103.232	103.664 (−)
f_{27}	429.253	317.448 (+)	361.635	1580.608 (−)	343.048	3401.997 (−)
f_{28}	2191.477	2201.395 (≈)	2172.795	3022.791 (−)	2160.542	3342.343 (−)
f_{29}	734.014	835.183 (−)	728.071	8191.131 (−)	779.0	158781.8 (−)
f_{30}	3830.2	6290.104 (−)	3315.001	13266.051 (−)	3280.476	45954.35 (−)
$+/\approx/-$	-	2 / 2 / 4	-	0 / 0 / 8	-	0 / 0 / 8

cores for each GPU.

- According to the aggregative results of statistical analysis, $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ achieves significantly better computing speed than Single-$\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ without sacrificing the solution quality. It shows that the dual control mode does not bring any side effects on solution quality.

7.3.5 Performance Analysis on Dynamic Regrouping Strategy

In order to investigate whether the dynamic regrouping strategy can improve the solution quality of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$, we implement a variant of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ with a static single group (denoted as Static-$\text{SPEO}_{\text{HPC}_{\text{gpu}}}$), in which the dynamic regrouping is disabled and all individuals in the island interact with each other as traditional EAs. Table 7.6 presents the comparisons of FEVs obtained by $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ and Static-$\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ on 64 GPUs with three large island sizes ($N_s = 1024, 2048$ and 4096). The statistical results indicate that the regrouping strategy significantly improves the solution quality of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ especially with a larger island size. For example, when island size is 1024, $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ outperforms on 4 test problems and

Table 7.7: Unit price and maximal computing time with different budgets (1, 10, 50 and 100 USD) on AWS EC2.

#Device	Price / hour	Execution Time			
		1 USD	10 USD	50 USD	100 USD
2 GPUs	1.80 USD	00:33:20	05:33:20	27:46:40	55:33:20
16 GPUs	14.4 USD	00:04:10	00:41:40	03:28:20	06:56:40
64 GPUs	57.6 USD	00:01:02	00:10:25	00:52:05	01:44:10
32 CPU cores	1.36 USD	00:44:07	07:21:10	36:45:50	73:31:40
128 CPU cores	5.44 USD	00:11:01	01:50:10	09:10:50	18:21:40
512 CPU cores	21.76 USD	00:02:45	00:27:30	02:17:30	04:35:00

performs similarly at 2 problems; while $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ outperforms Static-$\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ on all 8 test problems if the island size increases to 2048 or 4096.

An interesting phenomenon also can be observed that Static-$\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ achieves much worse solution with a larger island size, while $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ achieves better solution quality even with an increasing island size. For example, Static-$\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ achieves the FEVs of f_{29} is 835.183 when island size is 1024, but it reaches to 8191.131 and 158781.8 when the island size increases to 2048 and 4096, respectively. It could be due to the fact that a larger population requires more FEs to converge despite that it has a better exploration ability.

In conclusion, the dynamic regrouping strategy plays a vital role on guaranteeing the adequate search behavior. Moreover, it also contributes to the better computational efficiency by getting rid of worries of weak convergence by a large island size.

7.3.6 Discussion on Cost-effectiveness

Although $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ can achieve significant speedups, it is true that GPU is charged more expensively than traditional CPU-based computing devices on commercial or academic parallel computing facilities. Table 7.7 presents the unit price and total runtime based on the pricing policy of CPU (C5 instance) and GPU (Nvidia k80 at g2 instance) on Amazon EC2. In order to demonstrate the excellent cost-effectiveness of the proposed $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$, we compare $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ with its CPU counterpart $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ and the state-of-the-art CPU-based CloudDE using the same computing budget. Specifically, the stop criterion is no longer a fixed

number of FEs, instead, they stop when a fixed computing budget runs out. As a result, the cost-effectiveness can be fairly evaluated because the more powerful and expensive device is only allowed a shorter runtime. For example, as the computing budget is equally assigned to 8 test functions with 15 independent runs, 64 GPUs with 100 USD only allows 01:44:10 (6250 seconds) total runtime and 52.08 seconds ($\frac{6250}{8*15} = 52.08$) for each run; however, 32 CPU cores with 100 USD can allow 73:31:40 (264700 seconds) total runtime and thus 2205.83 seconds for each run.

The $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ is executed on NCI with default configurations presented at Table 7.1 using the island size $N_s = 4096$, $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ is configured based on Chapter 4 using the global population size $NP = 8192$ and CloudDE is configured with default values at work [36] using the global population size $NP = 8192$. Table 7.8, 7.9, 7.10 and 7.11 show the solution quality obtained by three algorithms with four budgets of 1, 10, 50 and 100 USD. Here, the proposed $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ is executed on 2, 16 and 64 GPUs, the $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ and CloudDE are executed on up to 512 CPU cores. According to these tables, we can have following observations:

- Given any budget, $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ always achieves better solutions than $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ and CloudDE. It verifies the effectiveness of the proposed algorithm because it achieves significantly better solutions than CPU-based parallel EAs. Moreover, its efficiency is also proven since $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ always require less computing time according to Table 7.7. It is because the $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ runs significant faster and achieves far more FEs than two CPU-based parallel EAs even a shorter time is provided. As a result, $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ acquires significantly better solutions in a shorter time.

- Given a small budget, $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ performs better on a small number of GPUs than it on a large number of GPUs. For example, $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ on 2 GPUs performs the best when budget is 1 USD. The reason is that $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ on 64 GPUs has a very large global population and can not converge with insufficient FEs.

- Given a sufficient budget, a large number of GPUs benefit more to $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$. For example, $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ on 64 GPUs performs best when budget is 10, 50 or 100 USD.

Table 7.8: Comparison of mean FEVs of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ (2, 16 and 64 GPUs) with $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ (32, 128 and 512 cores) and CloudDE (32 cores) with 1 USD budget.

func	$\text{SPEO}_{\text{HPC}_{\text{gpu}}}$			$\text{SPEO}_{\text{HPC}_{\text{cpu}}}$			CloudDE
	2 GPUs	16 GPUs	64 GPUs	32 cores	128 cores	512 cores	32 cores
f_{23}	**348.235**	348.236	352.562	509.602	629.001	1573.255	348.243
f_{24}	373.604	**369.117**	391.502	390.091	397.11	578.667	382.395
f_{25}	**200.0**	200.013	238.057	248.206	242.034	374.876	293.687
f_{26}	**103.28**	**103.267**	103.327	195.266	205.974	333.87	186.774
f_{27}	**420.978**	582.972	1841.231	2332.742	3594.973	4626.359	2321.961
f_{28}	**2248.018**	2299.403	3362.088	4567.029	10270.642	20214.3	2529.755
f_{29}	**734.964**	2184.698	36578.001	1829360.0	$4.37*10^7$	$1.54*10^9$	5678.327
f_{30}	**4477.952**	6740.112	58968.664	134219.18	616908.933	$1.24*10^7$	38471.967

It is because a large number of GPUs have a larger global population as well as better diversity, thus it requires a larger budget to converge.

- Given an increasing budget, $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ only improves slightly especially when the budget is increased from 50 to 100 USD. On the contrary, $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ and CloudDE can achieve remarkable improvements on solution quality. The reason is that $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ and CloudDE run slowly and require more time to achieve sufficient FEs to converge, while $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ converges much more quickly by acquiring sufficient FEs early due to the fast speed. Therefore, $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$is ideal to execute parallel EAs when the budget is very limited.

7.4 Conclusions

This chapter extends the $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ to the $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ which works efficiently on GPU-enabled HPC. In this work, we design a dual control mode to increase the GPU utilisation. We then implement the $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ with a GPU-based DE with a dynamic regrouping strategy over up to 64 GPUs. The results demonstrate that $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ outperforms $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ and a state-of-the-art EA in terms of solution quality, computational speed and cost-effectiveness.

Table 7.9: Comparison of mean FEVs of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ (2, 16 and 64 GPUs) with $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ (32, 128 and 512 cores) and CloudDE (32 cores) with 10 USD budget.

func	$\text{SPEO}_{\text{HPC}_{\text{gpu}}}$			$\text{SPEO}_{\text{HPC}_{\text{cpu}}}$			CloudDE
	2 GPUs	16 GPUs	64 GPUs	32 cores	128 cores	512 cores	32 cores
f_{23}	**348.235**	**348.235**	**348.235**	424.284	356.557	442.171	**348.235**
f_{24}	373.604	367.528	365.197	385.395	366.73	**360.771**	379.055
f_{25}	**200.0**	**200.0**	**200.0**	244.376	211.692	219.001	257.911
f_{26}	**103.223**	**103.216**	**103.223**	200.94	193.75	188.849	211.588
f_{27}	420.753	372.25	**344.849**	1819.561	1334.734	2895.289	1381.57
f_{28}	2247.955	2179.033	**2132.26**	3631.716	2630.76	4921.191	2231.881
f_{29}	734.222	**720.857**	754.16	541127.82	5528.651	497951.533	3698.155
f_{30}	4477.158	**3329.362**	**3125.927**	73007.427	27309.407	58203.967	20796.233

Table 7.10: Comparison of mean FEVs of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ (2, 16 and 64 GPUs) with $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ (32, 128 and 512 cores) and CloudDE (32 cores) with 50 USD budget.

func	$\text{SPEO}_{\text{HPC}_{\text{gpu}}}$			$\text{SPEO}_{\text{HPC}_{\text{cpu}}}$			CloudDE
	2 GPUs	16 GPUs	64 GPUs	32 cores	128 cores	512 cores	32 cores
f_{23}	**348.235**	**348.235**	**348.235**	395.74	348.239	348.684	**348.235**
f_{24}	373.604	367.528	365.197	385.238	364.394	**358.602**	381.523
f_{25}	**200.0**	**200.0**	**200.0**	237.197	201.943	200.339	244.519
f_{26}	**103.147**	103.165	103.189	200.769	200.207	186.886	192.577
f_{27}	420.753	372.25	**339.205**	1763.089	1133.897	1063.009	948.358
f_{28}	2247.955	2179.033	**2126.238**	3375.887	2662.148	2342.093	2197.535
f_{29}	734.222	720.671	**719.431**	448958.551	1491.095	1912.43	3178.194
f_{30}	4477.158	**3329.28**	**2949.839**	54375.767	13847.883	13153.736	15614.813

Table 7.11: Comparison of mean FEVs of $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ (2, 16 and 64 GPUs) with $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ (32, 128 and 512 cores) and CloudDE (32 cores) with 100 USD budget.

func	$\text{SPEO}_{\text{HPC}_{\text{gpu}}}$			$\text{SPEO}_{\text{HPC}_{\text{cpu}}}$			CloudDE
	2 GPUs	16 GPUs	64 GPUs	32 cores	128 cores	512 cores	32 cores
f_{23}	**348.235**	**348.235**	**348.235**	381.157	348.236	348.236	**348.235**
f_{24}	373.604	367.528	365.197	386.933	365.787	**359.423**	379.418
f_{25}	**200.0**	**200.0**	**200.0**	235.756	201.924	**200.0**	241.656
f_{26}	**103.117**	103.15	103.168	194.042	200.152	186.855	198.016
f_{27}	420.753	372.25	**339.205**	1685.643	1192.022	987.043	1011.66
f_{28}	2247.955	2179.033	**2126.237**	3514.614	2464.135	2277.12	2221.516
f_{29}	734.222	720.671	**719.418**	387996.721	1380.203	1511.169	3826.463
f_{30}	4477.158	**3329.28**	**2949.733**	52409.16	9505.127	9073.679	13940.613

Chapter 8

Conclusions and Future Work

8.1 Conclusions

The increasingly complex and large-scale problems bring a rapidly rising searching space and quickly exceeds the searching capabilities of traditional EAs. In recent years, large populations enter people's view and are increasingly employed to solve difficult real-world problems. Thus, the needed computational budget for large populations in this scenario may get prohibitive such that most of the existing EAs implemented in a sequential way would become incompetent given practically reasonable computational budget.

In this thesis, we experimentally study the performance of EAs with a large population on solving complex and complicated problems in terms of solution quality. Specifically, we apply two state-of-the-art algorithms and three generic EAs on eight difficult composition problems to investigate the ability to search good solutions of EAs with a large population. Experiments show that EAs with a large population can achieve significantly better solutions than those of EAs with a small population, as well as better speedups when implemented in parallel.

We also propose the $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ framework based on CPU-only HPC. This framework employ a buffer-based asynchronous migration strategy to improve the scalability of the proposed framework. We then implement this framework with a standard DE algorithm and an improved dynamic topology for information exchange on up to 512 CPU cores. The results present that $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ not only increases the computational efficiency but also improves the

solution quality when compared to a state-of-the-art island-based parallel EA.

Inspired by the extensive historical information produced by the parallel DE implemented based on $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$, we propose the LES-CDE which can use historical search information to improve the searching capability. Specifically, we design an ensemble of several neighbouring local models that are trained by OS-ELM to guide the generation of promising trial vectors. We also implement the LES-CDE in parallel based on the $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ framework. Results demonstrate that LES-CDE can significantly improve the search behavour of crowding DE and can be remarkably accelerated when it is implemented in parallel.

Motivated by the significant computing power of GPU, this thesis studies how to verify the correctness of implementations of GPU-based EAs on a single GPU. Specifically, we present an example that indicates the significance of correctness verification for GPU-based EAs. Then some GPU-inherent issues, which influence the output of GPU-based EAs including the library functions, the numerical precision, and the race condition, are discussed one by one. To cope with the issues mentioned above, a set of guidance is proposed to verify the correctness of the GPU-based EAs. An working example is presented in this chapter to examine the effectiveness of guidelines.

As a single GPU offers limited scalability, we further design the $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ which works efficiently on on-demand GPU devices at GPU-enabled HPC. In this work, we design a dual control mode to improve the scalability when increasing number of GPUs are demanded. We then implement the $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ with a GPU-based DE on up to 64 GPUs. The results demonstrate that $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ outperforms $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ and a state-of-the-art EA in terms of solution quality, computational speed and cost-effectiveness.

8.2 Future Work

There are much future work can be done to improve the research of this thesis. Some possible improvements are suggested as follows:

- The benefits of a large population can be examined on other difficult problems including some real engineering optimisation or benchmark functions. Moreover, some other famous

or state-of-the-art algorithms can be selected as the test algorithms. On the other hand, the relationship between the population size and reasonable FEs is also worth to be investigated. Such findings could be very important for researchers to decide how many parallel computing resources are necessary for specific large population size.

- The $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ framework employs a dynamic migration topology which randomly selects recipients among all islands. Thus, one improvement is to design a self-adaptive topology to further improve the information exchange between islands. The expected self-adaptive topology can select recipient islands based on their current status. For example, if a donor island finds a new solution that is very promising and unique, it will broadcast this solution to all other islands; if it always generates similar or bad solutions, the export of this island will be suppressed to avoid the occupancy of the communication system until some good solutions are found.

- As a simple and representative EA, the DE algorithm is mostly employed in this thesis to implement the $\text{SPEO}_{\text{HPC}_{\text{cpu}}}$ and $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ frameworks. Thus, one further improvement is to select some novel and state-of-the-art algorithms to examine the effectiveness of the two proposed frameworks. Another possible improvement is the employment of different EA operators and/or parameters for different islands simultaneously, so that complex optimisation problems can be better solved by various search patterns of different islands.

- The proposed parallel LES-CDE utilises one CPU core for each island to perform EA operations and train surrogate models. Consequently, this CPU core is easily occupied by the training tasks and the EA operations have to queue most of the time. Thus, one further improvement is to allocate some extra CPU cores to train surrogate models separately with the EA operations. In this way, computing resources can be allocated in a more flexible and efficient way; for example, more CPU cores can be assigned to perform training tasks which are usually more time-consuming than EA operations.

- The proposed $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ does not consider any direct communication between GPUs. However, using PCI-E system interconnect to solve large problems may be limited by the

bandwidth of PCI-E which increasingly becomes the bottleneck at the multi-GPU system level and drives the need for a faster and more scalable multiprocessor interconnect. Recently, Nvidia NVLink technology addresses this interconnect issue by providing higher bandwidth, more links, and improved scalability for multi-GPU and multi-GPU/CPU system configurations. Thus, one further improvement is to improve the framework to take advantage of this state-of-the-art technique and to improve communication efficiency.

- The GPU-enabled HPC utilised in this thesis comprises Nvidia Tesla K80 which was released in 2014. In these years, GPU devices develop rapidly, and four generations of GPU architectures (Maxwell, Pascal, Volta and Turing) are proposed since 2014. Thus, the performance of the proposed $\text{SPEO}_{\text{HPC}_{\text{gpu}}}$ framework can be better evaluated if it can be implemented and executed on various GPU-enabled HPC platforms with different GPUs.

Bibliography

[1] Thomas Bäck, David B Fogel, and Zbigniew Michalewicz. *Handbook of evolutionary computation*. CRC Press, 1997.

[2] Christian Blum and Xiaodong Li. *Swarm intelligence in optimization*. Springer, 2008.

[3] Kenneth A De Jong. *Evolutionary computation: a unified approach*. MIT press, 2006.

[4] James Kennedy, James F Kennedy, Russell C Eberhart, and Yuhui Shi. *Swarm intelligence*. Morgan Kaufmann, 2001.

[5] Carlos Groba, Antonio Sartal, and Xosé H Vázquez. Solving the dynamic traveling salesman problem using a genetic algorithm with trajectory prediction: An application to fish aggregating devices. *Computers & Operations Research*, 56:22–32, 2015.

[6] Runliang Dou, Chao Zong, and Minqiang Li. An interactive genetic algorithm with the interval arithmetic based on hesitation and its application to achieve customer collaborative product configuration design. *Applied Soft Computing*, 38:384–394, 2016.

[7] Enying Li and Hu Wang. An alternative adaptive differential evolutionary algorithm assisted by expected improvement criterion and cut-hdmr expansion and its application in time-based sheet forming design. *Advances in Engineering Software*, 97:96–107, 2016.

[8] Anton Bouter, Tanja Alderliesten, Arjan Bel, Cees Witteveen, and Peter AN Bosman. Large-scale parallelization of partial evaluations in evolutionary algorithms for real-world problems. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1199–1206. ACM, 2018.

[9] Roberto Santana, Pedro Larrañaga, and Jose A Lozano. Protein folding in simplified models with estimation of distribution algorithms. *IEEE transactions on Evolutionary Computation*, 12(4):418–438, 2008.

[10] Chen Jin, Yan-bo Zhu, Jing Fang, and Yi-tong Li. An improved methodology for arn crossing waypoints location problem. In *Digital Avionics Systems Conference (DASC), 2012 IEEE/AIAA 31st*, pages 4A5–1. IEEE, 2012.

[11] Heinz Mühlenbein, M Schomisch, and Joachim Born. The parallel genetic algorithm as function optimizer. *Parallel computing*, 17(6-7):619–632, 1991.

[12] Heinz Mühlenbein. Evolution in time and space–the parallel genetic algorithm. In *Foundations of genetic algorithms*, volume 1, pages 316–337. Elsevier, 1991.

[13] Heinz Mühlenbein, Martina Gorges-Schleuter, and Ottmar Krämer. Evolution algorithms in combinatorial optimization. *Parallel computing*, 7(1):65–85, 1988.

[14] Heinz Mühlenbein. Parallel genetic algorithms, population genetics and combinatorial optimization. In *Workshop on Parallel Processing: Logic, Organization, and Technology*, pages 398–406. Springer, 1989.

[15] Yue-Jiao Gong, Wei-Neng Chen, Zhi-Hui Zhan, Jun Zhang, Yun Li, Qingfu Zhang, and Jing-Jing Li. Distributed evolutionary algorithms and their models: A survey of the state-of-the-art. *Applied Soft Computing*, 34:286–300, 2015.

[16] Yang Wang, Yangyang Li, Zhenghan Chen, and Yu Xue. Cooperative particle swarm optimization using mapreduce. *Soft Computing*, 21(22):6593–6603, 2017.

[17] Kumar Utkarsh, Anupam Trivedi, Dipti Srinivasan, and Thomas Reindl. A consensus-based distributed computational intelligence technique for real-time optimal control in smart distribution grids. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 1(1):51–60, 2017.

[18] O Tolga Altinoz and Kalyanmoy Deb. Late parallelization and feedback approaches for distributed computation of evolutionary multi-objective optimization algorithms. *Neural Computing and Applications*, 30(3):723–733, 2018.

[19] Erick Cantu-Paz. Designing efficient master-slave parallel genetic algorithms. 1997.

[20] Enrique Alba. *Parallel metaheuristics: a new class of algorithms*, volume 47. John Wiley & Sons, 2005.

[21] Sanaz Mostaghim, Jurgen Branke, Andrew Lewis, and Hartmut Schmeck. Parallel multi-objective optimization using master-slave model on heterogeneous resources. In *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*, pages 1981–1987. IEEE, 2008.

[22] Marc Dubreuil, Christian Gagné, and Marc Parizeau. Analysis of a master-slave architecture for distributed evolutionary computations. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 36(1):229–235, 2006.

[23] Darrell Whitley. An overview of evolutionary algorithms: practical issues and common pitfalls. *Information and software technology*, 43(14):817–831, 2001.

[24] Lourdes Araujo and Juan Julián Merelo. Diversity through multiculturality: Assessing migrant choice policies in an island model. *IEEE Transactions on Evolutionary Computation*, 15(4):456–469, 2011.

[25] Zbigniew Skolicki. An analysis of island models in evolutionary computation. In *Proceedings of the 7th annual workshop on Genetic and evolutionary computation*, pages 386–389. ACM, 2005.

[26] Enrique Alba and Bernabé Dorronsoro. *Cellular genetic algorithms*, volume 42. Springer Science & Business Media, 2009.

[27] Enrique Alba and Marco Tomassini. Parallelism and evolutionary algorithms. *IEEE transactions on evolutionary computation*, 6(5):443–462, 2002.

[28] Antonio J Nebro, Juan J Durillo, Francisco Luna, Bernabé Dorronsoro, and Enrique Alba. Mocell: A cellular genetic algorithm for multiobjective optimization. *International Journal of Intelligent Systems*, 24(7):726–746, 2009.

[29] Enrique Alba and Bernabé Dorronsoro. The exploration/exploitation tradeoff in dynamic cellular genetic algorithms. *IEEE transactions on evolutionary computation*, 9(2):126–142, 2005.

[30] Mitchell A Potter and Kenneth A De Jong. A cooperative coevolutionary approach to function optimization. In *International Conference on Parallel Problem Solving from Nature*, pages 249–257. Springer, 1994.

[31] Zhenyu Yang, Ke Tang, and Xin Yao. Large scale evolutionary optimization using cooperative coevolution. *Information Sciences*, 178(15):2985–2999, 2008.

[32] Kay Chen Tan, YJ Yang, and Chi Keong Goh. A distributed cooperative coevolutionary algorithm for multiobjective optimization. *IEEE Transactions on Evolutionary Computation*, 10(5):527–549, 2006.

[33] Yong Liu, Xin Yao, Qiangfu Zhao, and Tetsuya Higuchi. Scaling up fast evolutionary programming with cooperative coevolution. In *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*, volume 2, pages 1101–1108. Ieee, 2001.

[34] Jinwoo Kim, Minyoung Kim, Mark-Oliver Stehr, Hyunok Oh, and Soonhoi Ha. A parallel and distributed meta-heuristic framework based on partially ordered knowledge sharing. *Journal of Parallel and Distributed Computing*, 72(4):564–578, 2012.

[35] Yan Y Liu, Wendy K Tam Cho, and Shaowen Wang. Pear: a massively parallel evolutionary computation approach for political redistricting optimization and analysis. *Swarm and Evolutionary Computation*, 30:78–92, 2016.

[36] Zhi-Hui Zhan, Xiao-Fang Liu, Huaxiang Zhang, Zhengtao Yu, Jian Weng, Yun Li, Tianlong Gu, and Jun Zhang. Cloudde: A heterogeneous differential evolution algorithm and

its distributed cloud version. *IEEE Transactions on Parallel and Distributed Systems*, 28(3):704–716, 2017.

[37] César Manuel Vargas Benítez and Heitor Silvério Lopes. A parallel genetic algorithm for protein folding prediction using the 3d-hp side chain model. In *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*, pages 1297–1304. IEEE, 2009.

[38] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Quoc Le, and Alex Kurakin. Large-scale evolution of image classifiers. *arXiv preprint arXiv:1703.01041*, 2017.

[39] Tim Salimans, Jonathan Ho, Xi Chen, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

[40] Daniel Leal Souza, Glauber Duarte Monteiro, Tiago Carvalho Martins, Victor Alexandrovich Dmitriev, and Otávio Noura Teixeira. Pso-gpu: accelerating particle swarm optimization in cuda-based graphics processing units. In *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, pages 837–838. ACM, 2011.

[41] Luca Mussi, Fabio Daolio, and Stefano Cagnoni. Evaluation of parallel particle swarm optimization algorithms within the cuda? architecture. *Information Sciences*, 181(20): 4642–4657, 2011.

[42] Laurence Dawson and Iain Stewart. Improving ant colony optimization performance on the gpu using cuda. In *Evolutionary Computation (CEC), 2013 IEEE Congress on*, pages 1901–1908. IEEE, 2013.

[43] You Zhou and Ying Tan. Particle swarm optimization with triggered mutation and its implementation based on gpu. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 1–8. ACM, 2010.

[44] You Zhou and Ying Tan. Gpu-based parallel particle swarm optimization. In *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*, pages 1493–1500. IEEE, 2009.

[45] Ying Tan and Ke Ding. A survey on gpu-based implementation of swarm intelligence algorithms. *IEEE transactions on cybernetics*, 46(9):2028–2041, 2016.

[46] Kenneth A De Jong and William M Spears. An analysis of the interacting roles of population size and crossover in genetic algorithms. In *International Conference on Parallel Problem Solving from Nature*, pages 38–47. Springer, 1990.

[47] Rainer Storn and Kenneth Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4): 341–359, 1997.

[48] Maurice Clerc and James Kennedy. The particle swarm-explosion, stability, and convergence in a multidimensional complex space. *IEEE transactions on Evolutionary Computation*, 6(1):58–73, 2002.

[49] S Ivvan Valdez, Arturo Hernández, and Salvador Botello. A boltzmann based estimation of distribution algorithm. *Information Sciences*, 236:126–137, 2013.

[50] Xianneng Li, Shingo Mabu, and Kotaro Hirasawa. A novel graph-based estimation of the distribution algorithm and its extension using reinforcement learning. *IEEE Trans. Evolutionary Computation*, 18(1):98–113, 2014.

[51] H-G Beyer and Bernhard Sendhoff. Evolution strategies for robust optimization. In *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, pages 1346–1353. Citeseer, 2006.

[52] Nacim Belkhir, Johann Dréo, Pierre Savéant, and Marc Schoenauer. Parameter setting for multicore cma-es with large populations. In *International Conference on Artificial Evolution (Evolution Artificielle)*, pages 109–122. Springer, 2015.

[53] Nikolaus Hansen and Stefan Kern. Evaluating the cma evolution strategy on multimodal test functions. In *International Conference on Parallel Problem Solving from Nature*, pages 282–291. Springer, 2004.

[54] Chang Wook Ahn and Rudrapatna S Ramakrishna. A genetic algorithm for shortest path routing problem and the sizing of populations. *IEEE transactions on evolutionary computation*, 6(6):566–579, 2002.

[55] Francisco Fernández de Vega, Erick Cantu-Paz, Jose I Lopez, and Tomas Manzano. Saving resources with plagues in genetic algorithms. In *PPSN*, pages 272–281. Springer, 2004.

[56] George Harik, Erick Cantú-Paz, David E Goldberg, and Brad L Miller. The gambler's ruin problem, genetic algorithms, and the sizing of populations. *Evolutionary Computation*, 7(3):231–253, 1999.

[57] Tobias Storch and Ingo Wegener. Real royal road functions for constant population size. *Theoretical Computer Science*, 320(1):123–134, 2004.

[58] Thomas Weise, Yuezhong Wu, Raymond Chiong, Ke Tang, and Jörg Lässig. Global versus local search: the impact of population sizes on evolutionary algorithm performance. *Journal of Global Optimization*, 66(3):511–534, 2016.

[59] Carsten Witt. Population size versus runtime of a simple evolutionary algorithm. *Theoretical Computer Science*, 403(1):104–120, 2008.

[60] Tianshi Chen, Ke Tang, Guoliang Chen, and Xin Yao. A large population size can be unhelpful in evolutionary algorithms. *Theoretical Computer Science*, 436:54–70, 2012.

[61] Jens Jagerskupper and Tobias Storch. When the plus strategy outperforms the comma strategyand when not. In *Foundations of Computational Intelligence, 2007. FOCI 2007. IEEE Symposium on*, pages 25–32. IEEE, 2007.

[62] Dawei Li and Li Wang. A study on the optimal population size of genetic algorithm. *world congress on intelligent control and automation*, 4:3019–3021, 2002.

[63] Yiyuan Gong and Alex Fukunaga. Distributed island-model genetic algorithms using heterogeneous parameter settings. In *Evolutionary Computation (CEC), 2011 IEEE Congress on*, pages 820–827. IEEE, 2011.

[64] L Johan Berntsson and Maolin Tang. A convergence model for asynchronous parallel genetic algorithms. 2003.

[65] Andrea Mambrini and Dario Izzo. Pade: A parallel algorithm based on the moea/d framework and the island model. In *International Conference on Parallel Problem Solving from Nature*, pages 711–720. Springer, 2014.

[66] Marek Ruciński, Dario Izzo, and Francesco Biscani. On the impact of the migration topology on the island model. *Parallel Computing*, 36(10-11):555–571, 2010.

[67] G. Folino, C. Pizzuti, and G. Spezzano. A scalable cellular implementation of parallel genetic programming. *IEEE Transactions on Evolutionary Computation*, 7(1):37–53, Feb 2003. ISSN 1089-778X. doi: 10.1109/TEVC.2002.806168.

[68] Pu Liu, Francis Lau, Michael J Lewis, and Cho-li Wang. A new asynchronous parallel evolutionary algorithm for function optimization. In *International Conference on Parallel Problem Solving from Nature*, pages 401–410. Springer, 2002.

[69] Yan Y Liu and Shaowen Wang. A scalable parallel genetic algorithm for the generalized assignment problem. *Parallel Computing*, 46:98–119, 2015.

[70] Irma R Andalon-Garcia and Arturo Chavoya. Performance comparison of three topologies of the island model of a parallel genetic algorithm implementation on a cluster platform. In *Electrical Communications and Computers (CONIELECOMP), 2012 22nd International Conference on*, pages 1–6. IEEE, 2012.

[71] Erick Cantu-Paz. *Efficient and accurate parallel genetic algorithms*, volume 1. Springer Science & Business Media, 2000.

[72] Markus Schwehm. Parallel population models for genetic algorithms. *Universität Erlangen-Nürnberg*, pages 2–8, 1996.

[73] Weihang Zhu. Massively parallel differential evolution—pattern search optimization with graphics hardware acceleration: an investigation on bound constrained optimization problems. *Journal of Global Optimization*, 50(3):417–437, 2011.

[74] Lucas de P Veronese and Renato A Krohling. Differential evolution algorithm on the gpu with c-cuda. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–7. IEEE, 2010.

[75] Mikhail Rabinovich, Phillip Kainga, David Johnson, Brandon Shafer, Jaehwan John Lee, and Rusell Eberhart. Particle swarm optimization on a gpu. In *Electro/Information Technology (EIT), 2012 IEEE International Conference on*, pages 1–6. IEEE, 2012.

[76] Vincent Roberge, Mohammed Tarbouchi, and Francis Okou. Strategies to accelerate harmonic minimization in multilevel inverters using a parallel genetic algorithm on graphical processing unit. *IEEE Trans. Power Electron*, 29(10):5087–5090, 2014.

[77] Shigeyoshi Tsutsui and Noriyuki Fujimoto. Aco with tabu search on a gpu for solving qaps using move-cost adjusted thread assignment. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 1547–1554. ACM, 2011.

[78] Yi Zhou, Fazhi He, and Yimin Qiu. Dynamic strategy based parallel ant colony optimization on gpus for tsps. *Science China Information Sciences*, 60(6):068102, 2017.

[79] Tsz Ho Wong, A Kai Qin, Shengchun Wang, and Yuhui Shi. cusade: A cuda-based parallel self-adaptive differential evolution algorithm. In *Proceedings of the 18th Asia Pacific Symposium on Intelligent and Evolutionary Systems-Volume 2*, pages 375–388. Springer, 2015.

[80] Miguel Lastra, Daniel Molina, and José M. Benítez. A high performance memetic algorithm for extremely high-dimensional problems. *Information Sciences*, 293:35–58, 2015. ISSN 00200255. doi: 10.1016/j.ins.2014.09.018. URL http://linkinghub.elsevier.com/retrieve/pii/S0020025514009244.

[81] Vijay Kalivarapu and Eliot Winer. A study of graphics hardware accelerated particle swarm optimization with digital pheromones. *Structural and Multidisciplinary Optimization*, 51(6):1281–1304, 2015.

[82] José M Cecilia, José M García, Andy Nisbet, Martyn Amos, and Manuel Ujaldón. Enhancing data parallelism for ant colony optimization on gpus. *Journal of Parallel and Distributed Computing*, 73(1):42–51, 2013.

[83] Pablo Vidal and Enrique Alba. A multi-gpu implementation of a cellular genetic algorithm. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–7. IEEE, 2010.

[84] Shigeyoshi Tsutsui and Noriyuki Fujimoto. On the effect of using multiple gpus in solving qaps with cuda. In *Proceedings of the 14th annual conference companion on Genetic and evolutionary computation*, pages 629–630. ACM, 2012.

[85] Tomáš Ježowicz, Petr Buček, Jan Platoš, and Václav Snášel. Evolutionary algorithms for fast parallel classification. In *Proceedings of the 9th International Conference on Computer Recognition Systems CORES 2015*, pages 659–670. Springer, 2016.

[86] Jiri Jaros. Multi-gpu island-based genetic algorithm for solving the knapsack problem. In *Evolutionary Computation (CEC), 2012 IEEE Congress on*, pages 1–8. IEEE, 2012.

[87] Sungjoo Ha and Byung-Ro Moon. Fast knowledge discovery in time series with gpgpu on genetic programming. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1159–1166. ACM, 2015.

[88] JJ Liang, BY Qu, and PN Suganthan. Problem definitions and evaluation criteria for the cec 2014 special session and competition on single objective real-parameter numerical optimization. *Computational Intelligence Laboratory, Zhengzhou University, Zhengzhou China and Technical Report, Nanyang Technological University, Singapore*, 2013.

[89] James Kennedy. Particle swarm optimization. In *Encyclopedia of machine learning*, pages 760–766. Springer, 2011.

[90] John H Holland. Genetic algorithms. *Scientific american*, 267(1):66–73, 1992.

[91] Kenneth Price, Rainer M Storn, and Jouni A Lampinen. *Differential evolution: a practical approach to global optimization*. Springer Science & Business Media, 2006.

[92] Kelly Fleetwood. An introduction to differential evolution. In *Proceedings of Mathematics and Statistics of Complex Systems (MASCOS) One Day Symposium, 26th November, Brisbane, Australia*, pages 785–791, 2004.

[93] Kenneth V Price. Differential evolution: a fast and simple numerical optimizer. In *Fuzzy Information Processing Society, 1996. NAFIPS., 1996 Biennial Conference of the North American*, pages 524–527. IEEE, 1996.

[94] Jakob Vesterstrom and Rene Thomsen. A comparative study of differential evolution, particle swarm optimization, and evolutionary algorithms on numerical benchmark problems. In *IEEE Congress on Evolutionary Computation*, volume 2, pages 1980–1987, 2004.

[95] Swagatam Das and Ponnuthurai Nagaratnam Suganthan. Differential evolution: a survey of the state-of-the-art. *IEEE transactions on evolutionary computation*, 15(1):4–31, 2011.

[96] Vitaliy Feoktistov. *Differential evolution*. Springer, 2006.

[97] Rainer Storn. On the usage of differential evolution for function optimization. In *Biennial conference of the North American fuzzy information processing society (NAFIPS)*, volume 519. IEEE Berkeley, 1996.

[98] A Kai Qin, Vicky Ling Huang, and Ponnuthurai N Suganthan. Differential evolution algorithm with strategy adaptation for global numerical optimization. *IEEE transactions on Evolutionary Computation*, 13(2):398–417, 2009.

[99] Jingqiao Zhang and Arthur C Sanderson. Jade: adaptive differential evolution with optional external archive. *IEEE Transactions on evolutionary computation*, 13(5):945–958, 2009.

[100] A Kai Qin and Ponnuthurai N Suganthan. Self-adaptive differential evolution algorithm for numerical optimization. In *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, volume 2, pages 1785–1791. IEEE, 2005.

[101] Efrñn Mezura-Montes, Jesús Velázquez-Reyes, and Carlos A Coello Coello. A comparative study of differential evolution variants for global optimization. In *Proceedings of the*

8th annual conference on Genetic and evolutionary computation, pages 485–492. ACM, 2006.

[102] Yuhui Shi and Russell C Eberhart. Empirical study of particle swarm optimization. In *Evolutionary computation, 1999. CEC 99. Proceedings of the 1999 congress on*, volume 3, pages 1945–1950. IEEE, 1999.

[103] Russell Eberhart and James Kennedy. A new optimizer using particle swarm theory. In *Micro Machine and Human Science, 1995. MHS'95., Proceedings of the Sixth International Symposium on*, pages 39–43. IEEE, 1995.

[104] Yuhui Shi and Russell C Eberhart. Parameter selection in particle swarm optimization. In *International conference on evolutionary programming*, pages 591–600. Springer, 1998.

[105] Riccardo Poli, James Kennedy, and Tim Blackwell. Particle swarm optimization. *Swarm intelligence*, 1(1):33–57, 2007.

[106] Russ C Eberhart and Yuhui Shi. Comparing inertia weights and constriction factors in particle swarm optimization. In *Evolutionary Computation, 2000. Proceedings of the 2000 Congress on*, volume 1, pages 84–88. IEEE, 2000.

[107] Ioan Cristian Trelea. The particle swarm optimization algorithm: convergence analysis and parameter selection. *Information processing letters*, 85(6):317–325, 2003.

[108] Russell C Eberhart and Yuhui Shi. Comparison between genetic algorithms and particle swarm optimization. In *International conference on evolutionary programming*, pages 611–616. Springer, 1998.

[109] Peter J Angeline. Evolutionary optimization versus particle swarm optimization: Philosophy and performance differences. In *International Conference on Evolutionary Programming*, pages 601–610. Springer, 1998.

[110] Maurice Clerc. *Particle swarm optimization*, volume 93. John Wiley & Sons, 2010.

[111] Yuhui Shi and Russell Eberhart. A modified particle swarm optimizer. In *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on*, pages 69–73. IEEE, 1998.

[112] Lawrence Davis. Handbook of genetic algorithms. 1991.

[113] Darrell Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

[114] SN Sivanandam and SN Deepa. Genetic algorithm optimization problems. In *Introduction to Genetic Algorithms*, pages 165–209. Springer, 2008.

[115] Christopher R Houck, Jeff Joines, and Michael G Kay. A genetic algorithm for function optimization: a matlab implementation. *Ncsu-ie tr*, 95(09):1–10, 1995.

[116] Michael D Vose. *The simple genetic algorithm: foundations and theory*, volume 12. MIT press, 1999.

[117] David E Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. In *Foundations of genetic algorithms*, volume 1, pages 69–93. Elsevier, 1991.

[118] Mandavilli Srinivas and Lalit M Patnaik. Genetic algorithms: A survey. *computer*, 27 (6):17–26, 1994.

[119] Aki Sorsa, Riikka Peltokangas, and Kauko Leiviska. Real-coded genetic algorithms and nonlinear parameter identification. In *Intelligent Systems, 2008. IS'08. 4th International IEEE Conference*, volume 2, pages 10–42. IEEE, 2008.

[120] Yuhui Shi. Brain storm optimization algorithm. *IEEE Congress on Evolutionary Computation*, 6728(CEC):1–14, 2011.

[121] Zhihui Zhan, Jun Zhang, Yuhui Shi, and Hailin Liu. A modified brain storm optimization. pages 1–8, 2012.

[122] José Ignacio Hidalgo and Francisco Fernández. Balancing the computation effort in genetic algorithms. In *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, volume 2, pages 1645–1652. IEEE, 2005.

[123] Anne Auger and Nikolaus Hansen. A restart cma evolution strategy with increasing population size. In *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, volume 2, pages 1769–1776. IEEE, 2005.

[124] Karin Zielinski, Shyam Vudathu, and Rainer Laur. Influence of different deviations allowed for equality constraints on particle swarm optimization and differential evolution. *Nature Inspired Cooperative Strategies for Optimization (NICSO 2007)*, pages 249–259, 2008.

[125] David E Goldberg, Kalyanmoy Deb, and James H Clark. Genetic algorithms, noise, and the sizing of populations. *Urbana*, 51:61801, 1991.

[126] David E Goldberg. *The design of innovation: Lessons from and for competent genetic algorithms*, volume 7. Springer Science & Business Media, 2013.

[127] Thomas Jansen, Kenneth A De Jong, and Ingo Wegener. On the choice of the offspring population size in evolutionary algorithms. *Evolutionary Computation*, 13(4):413–440, 2005.

[128] Jonathan E Rowe and Dirk Sudholt. The choice of the offspring population size in the (1, λ) ea. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, pages 1349–1356. ACM, 2012.

[129] Christian Gießen and Carsten Witt. The interplay of population size and mutation probability in the (1+lambda) ea on onemax. *Algorithmica*, 78(2):587–609, 2017.

[130] Xiaofeng Qi and Francesco Palmieri. Theoretical analysis of evolutionary algorithms with an infinite population size in continuous space. part i: Basic properties of selection and mutation. *IEEE Transactions on Neural Networks*, 5(1):102–119, 1994.

[131] Joao Carlos Costa, Rui Tavares, and Agostinho Rosa. An experimental study on dynamic random variation of population size. In *IEEE SMC'99 Conference Proceedings. 1999 IEEE International Conference on Systems, Man, and Cybernetics (Cat. No. 99CH37028)*, volume 1, pages 607–612. IEEE, 1999.

[132] Ting Hu and Wolfgang Banzhaf. Nonsynonymous to synonymous substitution ratio ka/ks: Measurement for rate of evolution in evolutionary computation. In *International Conference on Parallel Problem Solving from Nature*, pages 448–457. Springer, 2008.

[133] Yu-Fan Tung and Tian-Li Yu. Theoretical perspective of convergence complexity of evolutionary algorithms adopting optimal mixing. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 535–542, New York, NY, USA, 2015. ISBN 978-1-4503-3472-3. doi: 10.1145/2739480.2754685.

[134] Tobias Friedrich, Pietro S. Oliveto, Dirk Sudholt, and Carsten Witt. Analysis of diversity-preserving mechanisms for global exploration*. *Evol. Comput.*, 17(4):455–476, December 2009. ISSN 1063-6560.

[135] Kenneth A. De Jong and William M. Spears. An analysis of the interacting roles of population size and crossover in genetic algorithms. In *Parallel Problem Solving from Nature*, pages 38–47, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[136] Y. Zhang, M. Sakamoto, and H. Furutani. Effects of population size and mutation rate on results of genetic algorithm. In *2008 Fourth International Conference on Natural Computation*, volume 1, pages 70–75, Oct 2008. doi: 10.1109/ICNC.2008.345.

[137] R. Mallipeddi and P. N. Suganthan. Empirical study on the effect of population size on differential evolution algorithm. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, pages 3663–3670, June 2008.

[138] Admir Barolli, Tetsuya Oda, Evjola Spaho, Leonard Barolli, Fatos Xhafa, and Makoto Takizawa. Impact of population size and number of generations on the performance of dense wmns. *2012 Seventh International Conference on Broadband, Wireless Computing, Communication and Applications*, pages 523–528, 2012.

[139] R. A. Sarker and M. F. A. Kazi. Population size, search space and quality of solution: an experimental study. In *The 2003 Congress on Evolutionary Computation, 2003. CEC '03.*, volume 3, pages 2011–2018 Vol.3, Dec 2003.

[140] Adam P. Piotrowski. Review of differential evolution population size. *Swarm and Evolutionary Computation*, 32:1 – 24, 2017. ISSN 2210-6502.

[141] G. Zhang, Xiao-Xia Liu, and T. Zhang. The impact of population size on the performance of ga. In *2009 International Conference on Machine Learning and Cybernetics*, volume 4, pages 1866–1870, July 2009.

[142] A. Hernandez-Aguirre, B. P. Buckles, and A. Martinez-Alcantara. The probably approximately correct (pac) population size of a genetic algorithm. In *Proceedings 12th IEEE Internationals Conference on Tools with Artificial Intelligence. ICTAI 2000*, pages 199–202, Nov 2000. doi: 10.1109/TAI.2000.889870.

[143] E. Belmont-Moreno. The role of mutation and population size in genetic algorithms applied to physics problems. *International Journal of Modern Physics C*, 12(09):1345–1355, 2001.

[144] Weishan Dong and Xin Yao. Unified eigen analysis on multivariate gaussian based estimation of distribution algorithms. *Information Sciences*, 178(15):3000–3023, 2008. ISSN 0020-0255. Nature Inspired Problem-Solving.

[145] Tomoaki Tatsukawa, Takeshi Watanabe, and Akira Oyama. Evolutionary computation for many-objective optimization problems using massive population sizes on the k supercomputer. In *Evolutionary Computation (CEC), 2016 IEEE Congress on*, pages 1139–1148. IEEE, 2016.

[146] G. Roy, H. Lee, J. L. Welch, Y. Zhao, V. Pandey, and D. Thurston. A distributed pool architecture for genetic algorithms. In *2009 IEEE Congress on Evolutionary Computation*, pages 1177–1184, May 2009. doi: 10.1109/CEC.2009.4983079.

[147] T. Desell, D. P. Anderson, M. Magdon-Ismail, H. Newberg, B. K. Szymanski, and C. A. Varela. An analysis of massively distributed evolutionary algorithms. In *IEEE Congress on Evolutionary Computation*, pages 1–8, July 2010.

[148] Gabriel Luque, Enrique Alba, and Bernabé Dorronsoro. An asynchronous parallel implementation of a cellular genetic algorithm for combinatorial optimization. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pages 1395–1402, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-325-9. doi: 10.1145/1569901.1570088. URL http://doi.acm.org/10.1145/1569901.1570088.

[149] Juan Carlos Fuentes Cabrera and Carlos A. Coello Coello. *Micro-MOPSO: A Multi-Objective Particle Swarm Optimizer That Uses a Very Small Population Size*, pages 83–104. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[150] Yi Hong, Sam Kwong, Qingsheng Ren, and Xiong Wang. Over-selection: An attempt to boost eda under small population size. In *2007 IEEE Congress on Evolutionary Computation*, pages 1075–1082, Sep. 2007. doi: 10.1109/CEC.2007.4424589.

[151] W. Ashlock. Using very small population sizes in genetic programming. In *2006 IEEE International Conference on Evolutionary Computation*, pages 319–326, July 2006. doi: 10.1109/CEC.2006.1688325.

[152] Jiandong Mao and Juan Li. Dust particle size distribution inversion based on the multi population genetic algorithm. *Terrestrial, Atmospheric and Oceanic Sciences*, 25:791, 12 2014. doi: 10.3319/TAO.2014.06.12.01(A).

[153] K. Y. Kok, P. Rajendran, R. Rainis, and W. M. M. Wan Ibrahim. Investigation on selection schemes and population sizes for genetic algorithm in unmanned aerial vehicle path planning. In *2015 International Symposium on Technology Management and Emerging Technologies (ISTMET)*, pages 6–10, Aug 2015. doi: 10.1109/ISTMET.2015.7358990.

[154] Jarmo Alander. On optimal population size of genetic algorithms. pages 65 – 70, 06 1992. ISBN 0-8186-2760-3. doi: 10.1109/CMPEUR.1992.218485.

[155] Randy Haupt. Optimum population size and mutation rate for a simple real genetic algorithm that optimizes array factors. volume 15, pages 1034 – 1037 vol.2, 02 2000. ISBN 0-7803-6369-8. doi: 10.1109/APS.2000.875398.

[156] Daniel Mora-Meliá, P Iglesias-Rey, F Martinez-Solano, and P Ballesteros-Pérez. Efficiency of evolutionary algorithms in water network pipe sizing. *Water Resources Management*, 08 2015. doi: 10.1007/s11269-015-1092-x.

[157] Ali Nodehi, Mohamad Tayarani, and Fariborz Mahmoudi. A novel functional sized population quantum evolutionary algorithm for fractal image compression. *2009 14th International CSI Computer Conference*, pages 564–569, 2009.

[158] N. Allias, M. N. M. M. Noor, M. N. Ismail, and K. d. Silva. A hybrid gini pso-svm feature selection: An empirical study of population sizes on different classifier. In *2013 1st International Conference on Artificial Intelligence, Modelling and Simulation*, pages 107–110, Dec 2013.

[159] Jingrui Zhang, Qinghui Tang, Yalin Chen, and Shuang Lin. A hybrid particle swarm optimization with small population size to solve the optimal short-term hydro-thermal unit commitment problem. *Energy*, 109:765 – 780, 2016. ISSN 0360-5442.

[160] Zbigniew Skolicki and Kenneth De Jong. The influence of migration sizes and intervals on island models. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pages 1295–1302. ACM, 2005.

[161] Darrell Whitley, Soraya Rana, and Robert B Heckendorn. Island model genetic algorithms and linearly separable problems. In *AISB International Workshop on Evolutionary Computing*, pages 109–125. Springer, 1997.

[162] Erick Cantú-Paz. Migration policies, selection pressure, and parallel evolutionary algorithms. *Journal of heuristics*, 7(4):311–334, 2001.

[163] Jing Tang, Meng-Hiot Lim, Yew-Soon Ong, and Meng Joo Er. Study of migration topology in island model parallel hybrid-ga for large scale quadratic assignment problems. In

Control, Automation, Robotics and Vision Conference, 2004. ICARCV 2004 8th, volume 3, pages 2286–2291. IEEE, 2004.

[164] René Michel and Martin Middendorf. An island model based ant system with lookahead for the shortest supersequence problem. In *International Conference on Parallel Problem Solving from Nature*, pages 692–701. Springer, 1998.

[165] Douglas Antony Louis Piriyakumar and Paul Levi. A new approach to exploiting parallelism in ant colony optimization. In *Micromechatronics and Human Science, 2002. MHS 2002. Proceedings of 2002 International Symposium on*, pages 237–243. IEEE, 2002.

[166] Chunmei Zhang, Jie Chen, and Bin Xin. Distributed memetic differential evolution with the synergy of lamarckian and baldwinian learning. *Applied Soft Computing*, 13(5): 2947–2959, 2013.

[167] G Galeano, F Femdndez, M Tomassini, and L Vanneschi. Studying the influence of synchronous and asynchronous parallel gp on programs length evolution. In *Evolutionary Computation, 2002. CEC'02. Proceedings of the 2002 Congress on*, volume 2, pages 1727–1732. IEEE, 2002.

[168] Enrique Alba and José M Troya. Analyzing synchronous and asynchronous parallel distributed genetic algorithms. *Future Generation Computer Systems*, 17(4):451–465, 2001.

[169] Marcus Märtens and Dario Izzo. The asynchronous island model and nsga-ii: study of a new migration operator and its performance. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 1173–1180. ACM, 2013.

[170] Shingo Kurose, Kunihito Yamamori, Masaru Aikawa, and Ikuo Yoshihara. Asynchronous migration for parallel genetic programming on a computer cluster with multi-core processors. *Artificial Life and Robotics*, 16(4):533–536, 2012.

[171] Dario Izzo, Marek Rucinski, and Christos Ampatzis. Parallel global optimisation meta-heuristics using an asynchronous island-model. In *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*, pages 2301–2308. IEEE, 2009.

[172] Weihang Zhu and Yaohang Li. Gpu-accelerated differential evolutionary markov chain monte carlo method for multi-objective optimization over continuous space. In *Proceedings of the 2nd workshop on Bio-inspired algorithms for distributed systems*, pages 1–8. ACM, 2010.

[173] Pavel Krömer, Václav Snášel, Jan Platoš, and Ajith Abraham. Many-threaded implementation of differential evolution for the cuda platform. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 1595–1602. ACM, 2011.

[174] A Kai Qin, Federico Raimondo, Florence Forbes, and Yew Soon Ong. An improved cuda-based implementation of differential evolution on gpu. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, pages 991–998. ACM, 2012.

[175] Pavel Krömer, Jan Platoš, Václav Snášel, and Ajith Abraham. A comparison of many-threaded differential evolution and genetic algorithms on cuda. In *Nature and Biologically Inspired Computing (NaBIC), 2011 Third World Congress on*, pages 509–514. IEEE, 2011.

[176] Jarosław Arabas, Ogier Maitre, and Pierre Collet. Parade: a massively parallel differential evolution template for easea. In *Swarm and Evolutionary Computation*, pages 12–20. Springer, 2012.

[177] Vincent Roberge and Mohammed Tarbouchi. Parallel particle swarm optimization on graphical processing unit for pose estimation. *WSEAS Trans. Comput*, 11(6):170–179, 2012.

[178] Javier Reguera-Salgado and Julio Martín-Herrero. High performance gcp-based particle swarm optimization of orthorectification of airborne pushbroom imagery. In *Geoscience*

and Remote Sensing Symposium (IGARSS), 2012 IEEE International, pages 4086–4089. IEEE, 2012.

[179] Jan Platos, Vaclav Snasel, Tomas Jezowicz, Pavel Kromer, and Ajith Abraham. A pso-based document classification algorithm accelerated by the cuda platform. In *Systems, Man, and Cybernetics (SMC), 2012 IEEE International Conference on*, pages 1936–1941. IEEE, 2012.

[180] Zheng Zhang, Hock Soon Seah, Chee Kwang Quah, and Jixiang Sun. Gpu-accelerated real-time tracking of full-body motion with multi-layer search. *IEEE Transactions on Multimedia*, 15(1):106–119, 2013.

[181] Marco S Nobile, Daniela Besozzi, Paolo Cazzaniga, Giancarlo Mauri, and Dario Pescini. A gpu-based multi-swarm pso method for parameter estimation in stochastic biological systems exploiting discrete-time target series. In *European Conference on Evolutionary Computation, Machine Learning and Data Mining in Bioinformatics*, pages 74–85. Springer, 2012.

[182] Bhanu Sharma, Ruppa K Thulasiram, and Parimala Thulasiraman. Portfolio management using particle swarm optimization on gpu. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pages 103–110. IEEE, 2012.

[183] Ray-Bing Chen, Dai-Ni Hsieh, Ying Hung, and Weichung Wang. Optimizing latin hypercube designs by particle swarm. *Statistics and computing*, 23(5):663–676, 2013.

[184] MP Wachowiak and AE Lambe Foster. Gpu-based asynchronous global optimization with particle swarm. In *Journal of Physics: Conference Series*, volume 385, page 012012. IOP Publishing, 2012.

[185] Shigeyoshi Tsutsui and Noriyuki Fujimoto. Solving quadratic assignment problems by genetic algorithms with gpu computation: a case study. In *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, pages 2523–2530. ACM, 2009.

[186] Laurence Dawson and Iain A Stewart. Accelerating ant colony optimization-based edge detection on the gpu using cuda. IEEE, 2014.

[187] Darren M Chitty. Improving the performance of gpu-based genetic programming through exploitation of on-chip memory. *Soft Computing*, 20(2):661–680, 2016.

[188] Mehdi Goli, John McCall, Christopher Brown, Vladimir Janjic, and Kevin Hammond. Mapping parallel programs to heterogeneous cpu/gpu architectures using a monte carlo tree search. In *2013 IEEE Congress on Evolutionary Computation*, pages 2932–2939. IEEE, 2013.

[189] Pablo Vidal and Enrique Alba. Cellular genetic algorithm on graphic processing units. In *Nature Inspired Cooperative Strategies for Optimization (NICSO 2010)*, pages 223–232. Springer, 2010.

[190] Keith R Jackson, Lavanya Ramakrishnan, Krishna Muriki, Shane Canon, Shreyas Cholia, John Shalf, Harvey J Wasserman, and Nicholas J Wright. Performance analysis of high performance computing applications on the amazon web services cloud. In *2nd IEEE international conference on cloud computing technology and science*, pages 159–168. IEEE, 2010.

[191] Rajkumar Buyya et al. High performance cluster computing: Architectures and systems (volume 1). *Prentice Hall, Upper SaddleRiver, NJ, USA*, 1:999, 1999.

[192] Georg Hager and Gerhard Wellein. *Introduction to high performance computing for scientists and engineers*. CRC Press, 2010.

[193] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott Kohn, Lois McInnes, Steve Parker, and Brent Smolinski. Toward a common component architecture for high-performance scientific computing. In *High Performance Distributed Computing, 1999. Proceedings. The Eighth International Symposium on*, pages 115–124. IEEE, 1999.

[194] Gordon Bell and Jim Gray. What's next in high-performance computing? *Communications of the ACM*, 45(2):91–95, 2002.

[195] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming. *Parallel Computing*, 38(8):391–407, 2012.

[196] Aaftab Munshi. The opencl specification. In *Hot Chips 21 Symposium (HCS), 2009 IEEE*, pages 1–314. IEEE, 2009.

[197] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3): 66–73, 2010.

[198] Benedict Gaster, Lee Howes, David R Kaeli, Perhaad Mistry, and Dana Schaa. *Heterogeneous computing with openCL: revised openCL 1.* Newnes, 2012.

[199] Kamran Karimi, Neil G Dickson, and Firas Hamze. A performance comparison of cuda and opencl. *arXiv preprint arXiv:1005.2581*, 2010.

[200] CUDA Nvidia. Nvidia cuda c programming guide. *Nvidia Corporation*, 120(18):8, 2011.

[201] CUDA Nvidia. Compute unified device architecture programming guide. 2007.

[202] C Cuda. Best practices guide. *Nvidia Corporation*, 2012.

[203] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming.* Addison-Wesley Professional, 2010.

[204] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. In *ACM SIGGRAPH 2008 classes*, page 16. ACM, 2008.

[205] Shane Cook. *CUDA programming: a developer's guide to parallel computing with GPUs.* Newnes, 2012.

[206] David Luebke. Cuda: Scalable parallel programming for high-performance scientific computing. In *Biomedical Imaging: From Nano to Macro, 2008. ISBI 2008. 5th IEEE International Symposium on*, pages 836–838. IEEE, 2008.

[207] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel computing experiences with cuda. *IEEE micro*, (4):13–27, 2008.

[208] David Kirk et al. Nvidia cuda software and gpu parallel computing architecture. In *ISMM*, volume 7, pages 103–104, 2007.

[209] Tianyi David Han and Tarek S Abdelrahman. hi cuda: a high-level directive-based language for gpu programming. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 52–61. ACM, 2009.

[210] Mugdha A Rane. Fast morphological image processing on gpu using cuda. *Pune: Department of Computer Engineering and Information Technology, College of Engineering*, 2013.

[211] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. Gpu cluster for high performance computing. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 47. IEEE Computer Society, 2004.

[212] John Nickolls and William J Dally. The gpu computing era. *IEEE micro*, 30(2), 2010.

[213] Stephen W Keckler, William J Dally, Brucek Khailany, Michael Garland, and David Glasco. Gpus and the future of parallel computing. *IEEE Micro*, (5):7–17, 2011.

[214] Jack Dongarra and Michael A Heroux. Toward a new metric for ranking high performance computing systems. *Sandia Report, SAND2013-4744*, 312:150, 2013.

[215] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. Deep learning with cots hpc systems. In *International Conference on Machine Learning*, pages 1337–1345, 2013.

[216] Nikolaus Hansen, Anne Auger, Steffen Finck, and Raymond Ros. *Real-parameter black-box optimization benchmarking 2010: Experimental setup.* PhD thesis, INRIA, 2010.

[217] Jane-Jing Liang, Ponnuthurai Nagaratnam Suganthan, and Kalyanmoy Deb. Novel composition test functions for numerical global optimization. In *Swarm Intelligence Symposium, 2005. SIS 2005. Proceedings 2005 IEEE*, pages 68–75. IEEE, 2005.

[218] Ryoji Tanabe and Alex S Fukunaga. Improving the search performance of shade using linear population size reduction. In *Evolutionary Computation (CEC), 2014 IEEE Congress on*, pages 1658–1665. IEEE, 2014.

[219] Janez Brest, Mirjam Sepesy Maučec, and Borko Bošković. Single objective real-parameter optimization: algorithm jso. In *Evolutionary Computation (CEC), 2017 IEEE Congress on*, pages 1311–1318. IEEE, 2017.

[220] Jason E Cook and Daniel R Tauritz. An exploration into dynamic population sizing. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 807–814. ACM, 2010.

[221] Fernando G Lobo and Cláudio F Lima. Revisiting evolutionary algorithms with on-the-fly population size adjustment. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1241–1248. ACM, 2006.

[222] Jason Teo. Exploring dynamic self-adaptive populations in differential evolution. *Soft Computing-A Fusion of Foundations, Methodologies and Applications*, 10(8):673–686, 2006.

[223] Janez Brest and Mirjam Sepesy Maučec. Population size reduction for the differential evolution algorithm. *Applied Intelligence*, 29(3):228–247, 2008.

[224] Brad L Miller, David E Goldberg, et al. Genetic algorithms, tournament selection, and the effects of noise. *Complex systems*, 9(3):193–212, 1995.

[225] Dinabandhu Bhandari, CA Murthy, and Sankar K Pal. Genetic algorithm with elitist model and its convergence. *International Journal of Pattern Recognition and Artificial Intelligence*, 10(06):731–747, 1996.

[226] Kenneth Alan De Jong. Analysis of the behavior of a class of genetic adaptive systems. 1975.

[227] Salvador Garca, Daniel Molina, Manuel Lozano, and Francisco Herrera. A study on the use of non-parametric tests for analyzing the evolutionary algorithms behavior a case study on the cec2005 special session on real parameter optimization. *Journal of Heuristics*, 15(6):617, 2009.

[228] Ronald W Morrison and Kenneth A De Jong. Measurement of population diversity. In *Artificial Evolution*, volume 2310, pages 31–41. Springer, 2001.

[229] David W Walker and Jack J Dongarra. Mpi: A standard message passing interface. *Supercomputer*, 12:56–68, 1996.

[230] Jack Dongarra, Dennis Gannon, Geoffrey Fox, and Ken Kennedy. The impact of multicore on computational science software. *CTWatch Quarterly*, 3(1):1–10, 2007.

[231] Jonathan R Clausen, Daniel A Reasor Jr, and Cyrus K Aidun. Parallel performance of a lattice-boltzmann/finite element cellular blood flow solver on the ibm blue gene/p architecture. *Computer Physics Communications*, 181(6):1013–1020, 2010.

[232] Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew. Extreme learning machine: theory and applications. *Neurocomputing*, 70(1-3):489–501, 2006.

[233] Guang-Bin Huang, Qin-Yu Zhu, Chee-Kheong Siew, et al. Extreme learning machine: a new learning scheme of feedforward neural networks. *Neural networks*, 2:985–990, 2004.

[234] Nan-Ying Liang, Guang-Bin Huang, Paramasivan Saratchandran, and Narasimhan Sundararajan. A fast and accurate online sequential learning algorithm for feedforward networks. *IEEE Transactions on neural networks*, 17(6):1411–1423, 2006.

[235] Salvador García, Daniel Molina, Manuel Lozano, and Francisco Herrera. A study on the use of non-parametric tests for analyzing the evolutionary algorithms' behaviour: a case study on the cec'2005 special session on real parameter optimization. *Journal of Heuristics*, 15(6):617, 2009.

[236] Joaquín Derrac, Salvador García, Daniel Molina, and Francisco Herrera. A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms. *Swarm and Evolutionary Computation*, 1(1):3–18, 2011.

[237] Phillipe Pereira, Higo Albuquerque, Hendrio Marques, Isabela Silva, Celso Carvalho, Lucas Cordeiro, Vanessa Santos, and Ricardo Ferreira. Verifying cuda programs using smt-based context-bounded model checking. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 1648–1653. ACM, 2016.

[238] Guodong Li and Ganesh Gopalakrishnan. Scalable smt-based verification of gpu kernel functions. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 187–196. ACM, 2010.

[239] Alan Leung, Manish Gupta, Yuvraj Agarwal, Rajesh Gupta, Ranjit Jhala, and Sorin Lerner. Verifying gpu kernels by test amplification. In *ACM SIGPLAN Notices*, volume 47, pages 383–394. ACM, 2012.

[240] Jin Chen and A. K. Qin. A gpu-based implementation of brain storm optimization. In *Evolutionary Computation*, 2017.

[241] Yuhui Shi et al. Particle swarm optimization: developments, applications and resources. In *evolutionary computation, 2001. Proceedings of the 2001 Congress on*, volume 1, pages 81–86. IEEE, 2001.

[242] Carlos M Fonseca, Peter J Fleming, et al. Genetic algorithms for multiobjective optimization: Formulationdiscussion and generalization. In *Icga*, volume 93, pages 416–423, 1993.

[243] Thomas Bäck and Frank Hoffmeister. Extended selection mechanisms in genetic algorithms. 1991.

[244] Nathan Bell and Jared Hoberock. Thrust: A productivity-oriented library for cuda. *GPU computing gems Jade edition*, 2:359–371, 2011.

[245] Charles AR Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[246] Richard Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.

[247] Marco Zagha and Guy E Blelloch. Radix sort for vector multiprocessors. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 712–721. ACM, 1991.

[248] Dan Zuras, Mike Cowlishaw, Alex Aiken, Matthew Applegate, David Bailey, Steve Bass, Dileep Bhandarkar, Mahesh Bhat, David Bindel, Sylvie Boldo, et al. Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.

[249] Stephen H Unger. Hazards, critical races, and metastability. *IEEE Transactions on Computers*, 44(6):754–768, 1995.

[250] Juan Rada-Vilela, Mengjie Zhang, and Winston Seah. A performance study on synchronous and asynchronous updates in particle swarm optimization. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 21–28. ACM, 2011.

[251] A Serani, M Diez, C Leotardi, D Peri, G Fasano, U Iemma, and Emilio F Campana. On the use of synchronous and asynchronous single-objective deterministic particle swarm optimization in ship design problems. In *1st International Conference on Engineering and Applied Sciences Optimization, Kos*, 2014.