PUCRS

ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

FILIPO NOVO MÓR

AN EVOLUTIONARY APPROACH FOR THE TASK MAPPING PROBLEM

Porto Alegre
2016

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica
do Rio Grande do Sul

Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Informática
Programa de Pós-Graduação em Ciência da Computação

AN EVOLUTIONARY APPROACH FOR THE TASK MAPPING PROBLEM

FILIPO NOVO MÓR

Dissertation submitted to the Pontifícia Universidade Católica do Rio Grande do Sul in partial fullfillment of the requirements for the degree of Master in Computer Science.

Advisor: Prof. Ph.D. César Augusto Missio Marcon
Co-Advisor: Prof. Ph.D. Andrew Rau-Chaplin

Porto Alegre
2016

Ficha Catalográfica

M827a Mor, Filipo Novo

An Evolutionary Approach for the Task Mapping Problem / Filipo Novo Mor . – 2016.
82.
Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. César Augusto Missio Marcon.
Co-orientador: Prof. Dr. Andrew Rau-Chaplin.

1. NOC. 2. Evolução Diferencial. 3. Mapeamento. 4. Algoritmos Evolucionários. 5. Tarefas. I. Marcon, César Augusto Missio. II. Rau-Chaplin, Andrew. III. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).
Bibliotecária responsável: Salete Maria Sartori CRB-10/1363

Filipo Novo Mór

An Evolutionary Approach for the Task Mapping Problem

This Dissertation has been submitted in partial fulfillment of the requirements for the degree of Master of Computer Science, of the Graduate Program in Computer Science, School of Computer Science of the Pontifícia Universidade Católica do Rio Grande do Sul.

Sanctioned on August, 18th, 2016.

COMMITTEE MEMBERS:

Prof. Dr. Ricardo Melo Czerkster (PPGSPI/UNISC)

Prof. Dr. Fabiano Passuelo Hessel (PPGCC/PUCRS)

Prof. Dr. César Augusto Missio Marcon (PPGCC/PUCRS - Advisor)

Prof. Dr. Andrew Rau-Chaplin (Dalhousie University – Co-advisor)

Uma abordagem Evolucionária para o Problema de Mapeamento de Tarefas em Redes em Chip

RESUMO

Este trabalho têm como objetivo a implementação de um algoritimo evolucionário, baseado no algoritmo de Evolução Diferencial (DE), para a resolução do problema de Mapeamento de Tarefas em Redes em Chip. Foi implementada uma variação do algoritmo clássico de Evolução Diferencial, alterando-se o procedimento de operação genética da etapa de Recombinação, que passou a premiar individuos com base na existencia de uma condição indicativa de maior proximidade entre tarefas muito comunicantes. Nossa implementação foi avaliada a partir do uso do pacote de benchmark *NASA Numerical Aerodynamic Simulation* (NASA NAS) e os resultados mostraram que nossa implementação do DE se mostrou viável e competitiva. Na comparação com o mapeamento realizado com o framework CAFES, nossa implementação se mostrou superior em duas das 5 aplicações testadas, obtendo desempenho equivalente ao CAFES em uma aplicação e obtendo soluções menos eficientes em duas aplicações.

Palavras-Chave: NoC, Evolução Diferencial, Tarefas, Mapeamento, Algoritmos Evolucionários.

An Evolutionary Approach For the Task Mapping Problem

ABSTRACT

This works has the goal to implement an Evolutionary Algorithm, based on the classical Differential Evolution, to solve the Task Mapping onto NoC problem. Our variant implemented a changing on the genetic operator of recombination, that started to reward individuals containing a pre-select condition that indicates when most communicating tasks are allocated near to each other onto the NoC. Our implementation was subject to the *NASA Numerical Aerodynamic Simulation* (NASA NAS) benchmark and results have shown that our variant is feasible and competitive. When compared to the CAFES Framework, our DE variant presented superior results on two of five tested applications, reaching equivalent quality on one of the applications and getting worst results in two of them.

Keywords: NoC, Differential Evolution, Task, Mapping, Evolutionary Algorithms.

List of Figures

List of Tables

List of Acronyms

ACA – Ant Colony Algorithm

AI – Artifical Inteligence

CDCG – Communication Dependence and Computation Graph

CAFES – Communication Analysis For Embedded Systems

CCCP – Capacitated Centered Clustering Problem

CEM – Cross-Entropy Method

CMP – Chip-Multiprocessors

CR – Crossover Rate

CRG – Communication Resource Graph

CT-GA – Core-Tile Mapping

CUDA – Compute Unified Device Architecture

CWG – Communication Weighted Graph

DE – Differential Evolution

EA – Evolutionary Algorithm

ES – Exhaustive Search

ESA – Evolutionary Strategies

FA – Firefly Algorithm

FFT – Fast Fourier Transform

GA – Genetic Algorithm

GPU – Graphic Processor Unit

HPC – High Performance Computing

IACA – Improved Ant Colony Algorithm

IBGA – Island Based Genetic Algorithm

IFFT – Inverse Fast Fourier Transform

MF – Mutation Factor

MODE – Multi-Objective Differential Evolution

MPSoC – Multi-Processor on Chip

NAS – Numerical Aerodynamic Simulation

NDPAD – Gaussian Process Model Assisted Differential Evolution

NoC – Network-On-Chip

NPB – Parallel Benchmarks

NSGA-II – Non-dominated Sorting Genetic Algorithm II

PE – Processing Element

PMAP – Physical Mapping Algorithm

PWM – Pulse Width Modulation

SA – Simulated Annealing

SBDE – Selection-Based Differential Evolution Algorithm

SIMD – Single Instruction, Multiple Data

SoC – System on Chip

SODE – Single-Objective Differential Evolution

SPEA2 – Strength Pareto Evolutionary Algorithm 2

TG – Task Graph

VOPD – Video Objective Plane Decoder

XML – Extensible Markup Language

Contents

1. INTRODUCTION

Modern computer systems are frequently used to process a large amount of data or to perform an enormous number of instructions. One of the most challenging problems currently tackled by computer systems is the Task Mapping problem, which consist, basically, in finding the best configuration for allocate tasks onto a target architecture, seeking to optimize this allocation accordingly to some objective criterion. For example, to reduce power consumption, or to reduce application execution time, are very common when seeking optimizations. However, considering that Task Mapping is an NP-Hard class problem, brute force-based methods are not feasible when the number of tasks is big, or even the number of Available Processing Elements (PEs) is high. Due to this, heuristic-based methods are preferably to find acceptable solutions in reasonable time, although there is no guarantee these solutions are the best existing possibilities for solving the problem. In this context, the family of searching techniques is huge, including mature and widely spread algorithms, such as the Simulated Annealing (SA) or the Differential Evolution (DE) algorithm.

One challenge is trying to keep a similar, or related a task, close to one another after mapped to the target architecture. Depending on the complexity of the system, or when the system runs more than one application at a time, the scenario may became too complex to traditional non-heuristic techniques to work efficiently. One application can be decomposed in different tasks, where each task is defined as an interdependent set of instructions and data that performs data processing and communication.

Network-On-Chips (NoCs) are communication infrastructures similar to computer networks. Although it may use the basic concepts of larger scaled computer networks, NoCs have particularities due to the fact they are implemented within a chip. NoCs normally are flexible to support concurrent communications, which make them suitable to implement applications that require a large amount of communicating tasks and are computing intensive [11].

Tests have shown that when compared to other heuristic methods, genetic algorithms provide the fastest way to solve the Task Mapping Problem, specially if the chromosome in the initial Population set represents a mapping solution from another mapping algorithm [7].

Evolutionary Algorithms (EA) are heuristic-based methods that implement the following procedures: variation operators (such as *Mutation* and *Recombination*) to create and keep diversity, and to expand the search through the space of solutions; and the selection operator, that acts to push the quality for the solutions.

The *Differential Evolution* (DE) algorithm is one of the most powerful stochastic real-parameter optimization algorithms in current use. DE operates through similar computational steps as employed by a standard EA. However, unlike traditional EAs, the DE-

variants perturb the current-generation population members with the scaled differences of randomly selected and distinct population members [13]. Because of the way it represents its individuals, DE is ideal for representing non-linear problems. For this reason we have chosen DE for develop this work.

In this work we propose a novel implementation of the DE, trying to enhance the capabilities of a common genetic operator to also account some characteristics that may raise to better off-springs faster.

1.1 Motivation

Many related works are based on traditional implementations of Genetic or Evolutionary algorithms. However, few works are grounded on modifications of the base strategies of these techniques. Our proposed implementation seeks to take advantage of the features of the classical DE algorithm, which allows to implement optimizations based on non-linear problems, and at the same time use the simplicity and flexibility of its code to reach a more efficient solver for the task mapping problem.

1.2 Objectives

This work aims to implement a efficient new approach of the DE algorith, in Single Objective Mode, to solve the Task Mapping in NoCs.

1.3 Document Outline

This document is organized as follows. Chapter 1 briefly introduces this work and how the document is organized. In Chapter 2, we cover all the theoretical concepts that were explored in this work, as well as a description of the benchmark applications that were used to evaluate the quality of our implementations. Chapter 3 describes the main related works and at the end, makes a brief comparison among these works and our proposal. At Chapter 4, Project Methodology, we describe our implementation, the metrics used and the tools we have developed: concurrently, the modification we executed on the genetic operator of the DE is described. Chapter 5 describes the test scenarios, and how the tests were executed and the obtained results. Finally, Chapter 6 presents our conclusions and final considerations.

2. THEORETICAL BACKGROUND

This chapter surveys two topics: the family of evolutionary algorithms, with a specific focus on the Differential Evolution algorithm and the task mapping for the NoC problem. As mentioned before, the DE algorithm is possibly one of the most robust stochastic real-parameter optimization algorithms currently in use [13]. At the same time, the task-mapping problem on NoC architectures is very challenging, especially when new tasks of different applications have to be supported at run-time [49].

2.1 Task Mapping Problem

Given the set of origin objects $X = \{x_1, x_2, ..., x_c\}$ and the set of destination objects $Y = \{y_1, y_2, ..., y_p\}$, mapping is a *complete injective function* $\varphi : X \rightarrow Y$ that associates the objects from the origin set to the destination set objects having $|Y| \geq |X|$. This association is called *map* [35].

For this work, the set of objects in the origin is formed by groups of tasks from a given application, while the NoC tiles form the set of destination objects. Although the mapping literature is vast, the mapping process may be classified according to the following four criteria: (i) the target architecture; (ii) the number of tasks per Processing Element (PE); (iii) the moment in which a task is executed; and (iv) system management approach [33].

According to the target architecture, task mapping can be performed in homogeneous (identical PEs) or heterogeneous (e.g. DSP, dedicated IPs, accelerators) systems. Regarding the number of tasks mapped per PEs, mapping approaches can be classified as single or multi-task. *Single-task* assumes only one task assignment per PE while *multi-task* allows mapping more than one task per PE according to some criteria (e.g. communication, execution time, task deadlines) [33].

The mapping process can be defined at design-time or run-time. When task mapping is defined at design-time (also referred as *offline* or *static mapping approach*), all applications that will be executed in the system must be known in advance [33].

This work focuses on mapping single-task and multi-task applications onto homogeneous and high-level architectures, specifically 2D NoC meshes. For multi-task mapping, the task grouping criterion used was the communication among tasks.

2.1.1 Partitioning versus Mapping

Figure 2.1 shows a sample application represented by three different levels (double border rectangles). In the first level, the application *App1* is compounded by a *task group* $T = \{t_1, t_2, ..., t_n\}$. A partitioning is performed over this application for grouping tasks according to some objective criterion; thus generating a *set of cores* $N = \{n_1, n_2, ..., n_c\}$ [35]. For this work, the objective of the partitioning is to reduce the volume of communication among tasks.

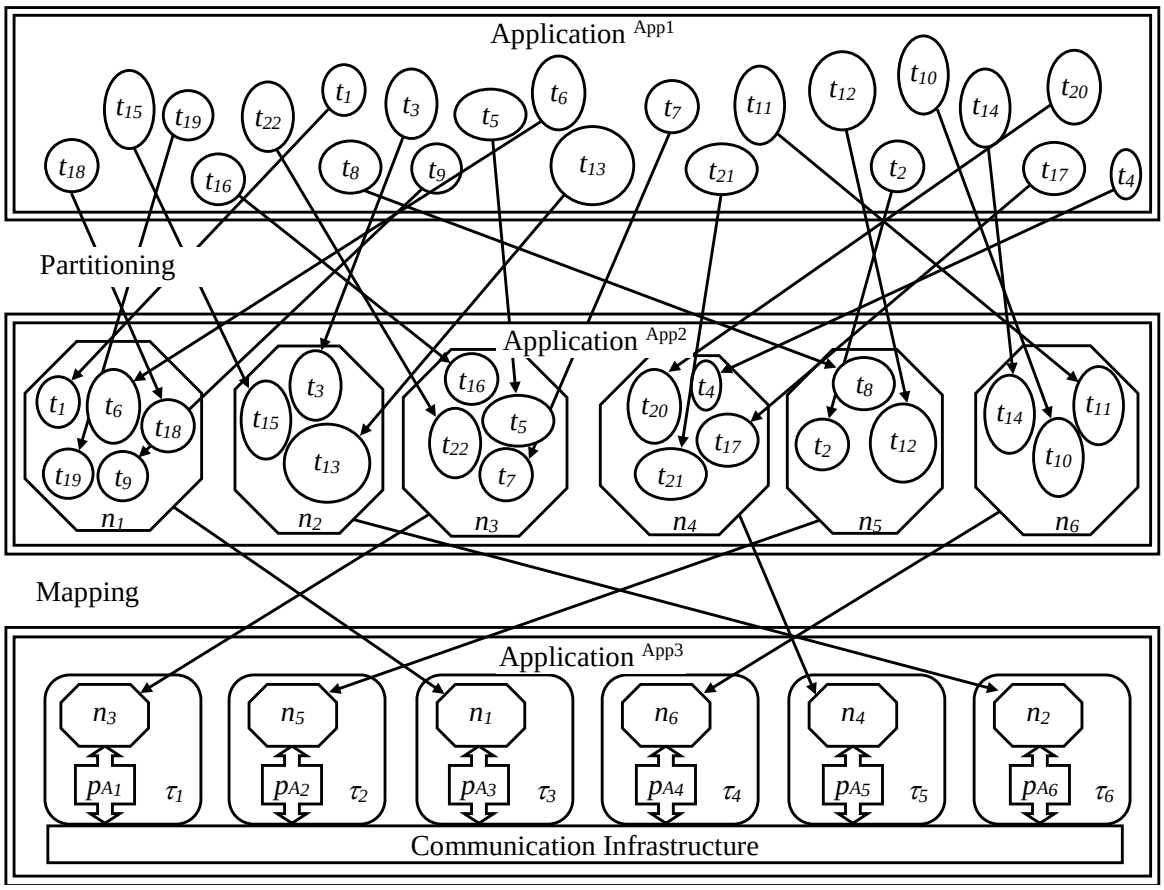


Figure 2.1: Partitioning and Mapping Process. By [35]

The second level shows *App2*, which is made of a set of cores. A set of tiles $\tau = \{t_1, t_2, ..., t_p\}$ behaves like a physical medium to position these cores to the communication infrastructure which is the compound of a set of *access points* $Pa = \{pa_1, pa_2, ..., pa_p\}$ [35].

Being that $O = \{o_1, o_2, ..., o_n\}$ is the set of all objects from a system, and $b \subseteq O$ is a subset of O called *block* and $P = \{b_1, b_2, ..., b_m\}$ is a partition formed by all blocks of O; a partitioning is the process that generates P, in a way that $b_1 \cup b_2 \cup ... \cup b_m = O$ and $b_k \cap b_i = \oslash \forall b_k, b_i$ having $k \neq 1$ [35].

While the partitioning has the intention to group tasks according to an objective criterion and expose opportunities for parallel execution [8], mapping has the goal of allocate

these groups to the PEs of the target architecture. Traditional non-evolutionary approaches tend to use different criteria for each step. For example, it might be possible to first group tasks with a high communication volume amongst them (partitioning), and only then map these groups onto the target architecture using an energy consumption criterion [35]. A fitness function may be used to evaluate the quality of resulting groups. It may or may not take into account information about the target architecture. For example, the distance between two tiles may profoundly impact the number of hops required for a message to be sent within the NoC. Therefore the fitness function should consider the distance when evaluating a candidate partitioning solution. The same may happen regarding mapping solutions.

2.1.2 Task Mapping Algorithms

Task mapping algorithms may be roughly classified as [8]: graph theoretic algorithms, mathematical programming, and heuristic algorithms. These algorithms will be briefly reviewed.

Graph-Theoretic Algorithms

Graph-Theoretic Algorithms frequently require a graph representing the application as an input. This graph may be generated by the partitioning step and must implement task groups according to specific criteria, such as execution time, message exchange or any other parameter [8]. Some algorithms from this family are the *Network Flow Algorithm* [24], which uses the *Max Flow/Min Cut* technique to find assignments and minimizes execution and communication costs [8]; the *Shortest Tree Algorithm*, which assigns modules (partitions) of a program to an non-homogeneous target architecture [6]; the *A* Algorithm*, that implements the *minimax* criterion, which is based on both minimization of inter-processor communication and processor load balance [48].

Mathematical Programming

Mathematical Programming uses an alternative approach to the task mapping problem. It considers the constraints of the target architecture, such as the capacity of the processing elements, memory capacity and also communication constraints (routing tables, shorter routes, etc.) [17]. These constraints are represented by mathematical inequality equations [9] [55].

Heuristic-based Algorithms

The fact that the task mapping problem is an NP-complete class problem, makes heuristic-based techniques very interesting to tackle. Any computing effort that skips brute force approaches will benefit from shorter execution times; although there is no guarantee that the best global solution will be found. Among the myriad of heuristic-based techniques, there is the family of Evolutionary Algorithms.

2.2 Evolutionary Algorithms

The history of Evolutionary Algorithms began during the 1940's, when scientists were inspired by using natural processes for the first time to help create the branch of AI (Artificial Intelligence) within the Computer Science discipline. Their research was initially developed with a focus on the working of cognitive and learning processes. Since then, the use of Evolutionary Algorithms have spread throughout the scientific community, raising a myriad of new applications, that have helped to solve many significant problems using very simple computational approaches [32].

Evolutionary Algorithms use computing models based on natural processes as a tool for solving problems. Although there are many different proposed computing models inside this area, they all have in common the concept of mimicking the natural selection process by implementing routines of selection, mutation and reproduction of individuals [3].

Algorithms from this family work by keeping a set of structures called population, which is formed by units named individuals, whose behaviour mimics the natural selection process. Each individual represents a possible solution for the problem. This representation is made up of a set of chromosomes, which, in turn, are responsible for the discretization of the problem according to the computing model being used. Each chromosome is associated with only one characteristic of the solution. Classic implementations usually use values '1' or '0' to indicate the presence (or absence) of the feature associated to the chromosome. However, it is possible to find implementations where the chromosomes may be valued using different ranges, such as real numbers (for example, from 0.00 to 1.00 and any real value in between) or even integer number (for example, from -5 to 10 and any integer value in between).

Figure 2.2 shows a graph representing a sample application. Each node represents one task, while each edge indicates messages sent from one task to another. The integer value associated with each edge informs the message size. For example, the task 'A' sends a message with size 5 to the task 'B'; the task 'B' send a message with size 3 to the task 'D' and so on.

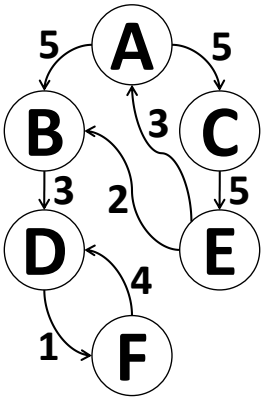


Figure 2.2: Graph representing a sample application.

The following example proposes to map the application from Figure 2.2 to the target architecture shown in Figure 2.3. This architecture is a 2D mesh (grid) formed by nine processing units arranged in three lines and three columns.

Each processing unit is identified by its location in the mesh by the pair $P = x, y$, where P is the position, x indicates the line and y indicates the respective column. Alternatively, a single integer number may identify a particular processing unit. This number is given by the formulae $P = (x * w) + y$, where P indicates the position, x indicates the line, w is the width (number of columns in the mesh), and y is the column identifier. This alternative indexing will be essential for later implementing the evolutionary algorithm. In this model, each processing unit may contain none or only one task associated at a time. For this reason, and as shown in this example, a maximum of nine tasks would be mapped. But since the sample application from Figure 2.2 includes only six tasks, a total of three processing units will remain idle after assigning those tasks to this 3x3 mesh.

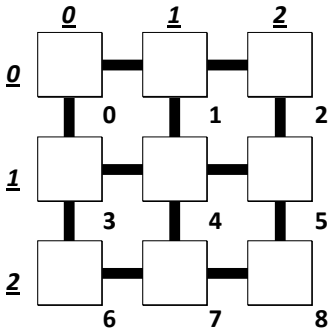


Figure 2.3: 3x3 2D MESH NoC

The population is compounded by different individuals which are randomly initialized. Each one of these individuals represent a proposed mapping of the sample application to the target architecture. For this reason, it is necessary to use a data structure which is able to indicate which task is associated with each processing unit. One possible solution is to use an array, where each array position represents one unique processing unit. When a task is mapped to a processing unit, it is possible to describe the association by filling the

array position corresponding to the chosen processing unit with the task identifier (in this case, this identifier is a letter, but typically numeric ID's are used for tasks), as in Figure 2.4.

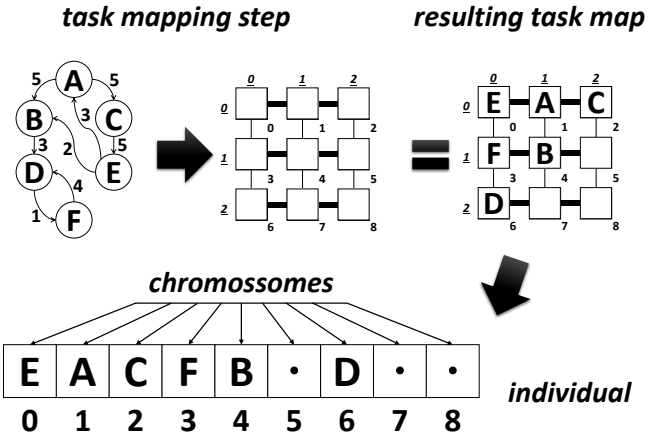


Figure 2.4: Representing a mapping solution as an individual.

In this example, the processing unit 0 (located at the line zero and column zero in the mesh) received the task 'E'. The processing unit 1 received the task 'A' and so on as shown in Figure 2.4.

A population is a group of individuals with different configurations. These individuals are initially set with random solutions, by simply generating random values for each one of its chromosomes.

Figure 2.5 shows an example of how a population can be structured. Each line represents a complete individual (identified by the list of vertically arranged numbers from 0 to 3), while each column (determined by the horizontal numbers from 0 to 8) describes one chromosome.

	0	1	2	3	4	5	6	7	8
0	E	A	C	F	B	•	D	•	•
1	B	•	•	A	C	•	D	F	E
2	•	C	•	A	B	E	D	•	F
3	F	D	A	•	•	•	B	E	C

Figure 2.5: Example of Population.

These individuals are subject to operations performed by *genetic operators* (such as *recombination* and *mutation*). Each individual in the population is evaluated by a *fitness function* (also called *objective function* by some authors), which has the purpose to measure the quality of the individual according to one particular criterion.

The evaluating method used by the fitness function is very dependent on the underlying problem and its computing model of choice. Although most evolutionary algorithm

solutions will share a common basic code structure, each problem will demand a very specific fitness function to evaluate the quality of the individuals in the population. For this work, we have developed two different fitness functions, as shown in the section Section 4.2.

The main idea behind the use of a fitness function is to provide a way to classify the individuals in the population and then identify which of them are the best solutions for solving the underlying problem. A fitness function should always result in a scalar value, as to make it possible to sort the individuals by using this value as a reference.

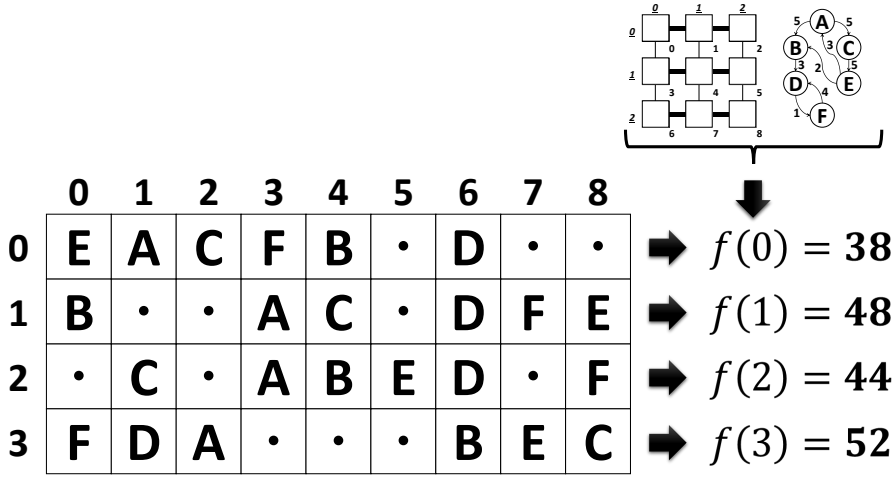


Figure 2.6: Population evaluation using a fitness function.

Figure 2.6 shows the basis of the evaluation process. The fitness function inputs include the target architecture, the application graph (in the example, representing the communication between tasks) and also the individual submitted to evaluation. In this example, for each individual, the fitness function will return one unique value representing the total communication volume. It will take into account the size of each message (from the application graph) and the number of hops taken from its origin in order to have reached its destination, and will have considered the task distribution described by the individual configuration. Thus, from the current generation, the best individual (considering the communication volume) will be the individual 0, followed by the individual 2, then followed by the individual 1 and finally by the individual 3.

2.2.1 Differential Evolution

The tree of Evolutionary Algorithms is formed by groups of different searching techniques, which are based on the exploration of the space of solutions and are grouped as a 'guided or random' branch. The main feature of the algorithms in this group is to perform a search that begins in a random way, but after some iterations, will start to follow towards a particular direction. According to [52], the strategy is to generate variations of the parameter vectors. Once a variation is generated, a decision must be made whether or not to accept

the newly derived parameters. Based on a greedy criterion, a new parameter vector is allowed if, and only if, it leads to a better value after being evaluated by the fitness function. This value may be required to increase or decrease, depending on the problem being optimized, and although it might allow the algorithm to converge reasonably fast, there is a risk of becoming trapped in a minimum local area, which will prevent the algorithm to reach the best existing solution.

When the fitness function represents a nonlinear and non-differentiable problem, direct search techniques are the methods of choice [52]. Some of these methods are *Genetic Algorithms* (GAs) [21], *Evolutionary Strategies* (ESAs) [44], *Simulated Annealing* (SA) [26] and the *Differential Evolution* (DE) [53]. Figure 2.7 displays an ontology with the relation among these techniques.

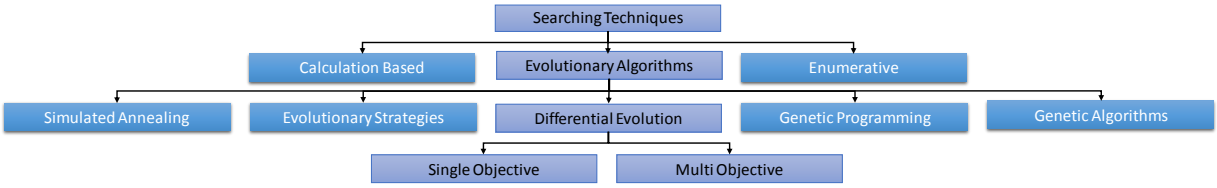


Figure 2.7: Family of Evolutionary Algorithms. Based on [32]

According to [52], users generally demand that a practical minimization technique should fulfill the following requirements:

(r.i) Ability to handle non-differentiable, nonlinear and multimodal fitness functions.

(r.ii) Parallelizability to cope with computation intensive fitness functions.

(r.iii) Ease of use. (i.e., limited control variables to steer the optimization. These variables should also be robust and easy to choose).

(r.iv) Good convergence properties. (i.e., consistent convergence to the global minimum in consecutive independent trials).

The DE was designed to be a stochastic direct search method. Direct search methods have the advantage of being easily applied to experimental optimizations where the cost value (calculated by the fitness function) is derived from a physical experimentation rather than from a computer simulation [53]. This makes the DE a good candidate to solve problems based on constraints like the requirement (r.i).

Depending on the problem being optimized, sometimes the fitness function might take from minutes to hours to complete the evaluation process of an individual. To obtain reasonable execution times, it may be interesting to try to parallelize the fitness function code.

The ability to divide the computing effort into different processors or threads is necessary when the fitness function might take many minutes to complete the evaluation

process of each individual on the population. Since the DE uses vector population, it is possible to evaluate each individual separately. This allows the DE to perfectly attend to the requirement (r.ii) [53].

Classical DE may be implemented with less than 10 lines of code. At the same time, there will be few controlling parameters, which will make the DE easy to use and simple to code [53].

The converge properties are mandatory for a good optimization algorithm, and the DE seems to reach good success rates when comparing to other similar techniques; as will be demonstrated in Section 5.1.3.

The DE is a direct search method which utilizes NP D-dimensional parameter vectors [52]:

$$x_{i,G},\ i = 1, 2, \ldots, NP \tag{2.1}$$

where G indicates the number of desired generations. NP does not change during execution time.

Figure 2.8 shows the main stages of the basic DE algorithm to be: *vector initialization*, *mutation*, *recombination*, and *selection*.

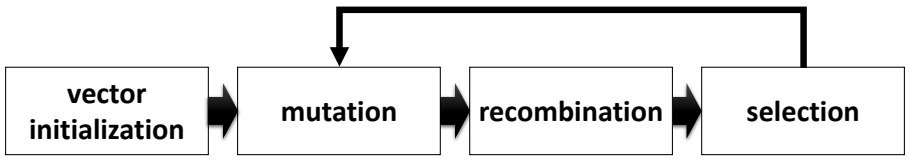


Figure 2.8: DE main stages.

During the *vector initialization phase*, the initial vector population is initialized randomly and must cover the entire parameter space [52]. It assumes a uniform probability distribution for all random decisions, and in the case that a preliminary solution is available, the initial population might be generated by adding normally distributed random deviations to the nominal solution $x_{nom,0}$ [52]. This operation is called *mutation* [52].

The *mutation* process consists of generating a new mutate vector for each target vector $x_{i,G},\ i = 1, 2, \ldots, NP$ as shown in (2.2):

$$V_{i,G+1} = X_{r^i_{1,G}} + F.(x_{r^i_{2,G}} - x_{r^i_{3,G}}) \tag{2.2}$$

During this process, a new parameter vector is generated by the DE by adding the weighted difference between two population vectors to a third vector α. This resulting vector is then scaled using the mutation factor F, generating a new target vector δ, as shown in Figure 2.9. This new vector will behave as the *donor vector* for the next phase. The mutation factor F is one of the execution parameters, and although it does not change

during executing time in the classical DE, there are alternative implementations where this factor is dynamically adjusted for reaching better results.

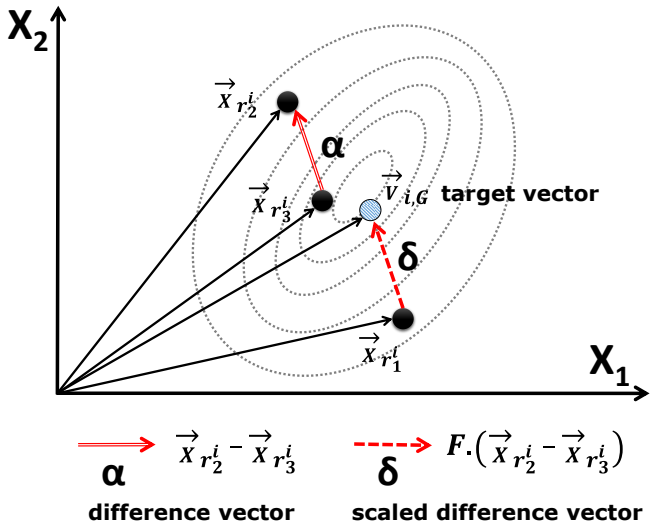


Figure 2.9: Mutation: an example of a two-dimensional fitness function showing its contour lines and the process for generating the trial parameter vector. Based on [13].

To enhance the potential diversity of the population, the *recombination operation* comes into play after generating the donor vector, which happens after mutation [13]. The donor vector exchanges its chromosomes with the current population vector $X_{i,G}$, forming the *trial vector* $U_{i,G} = [u_{1,i,G}, u_{2,i,G}, u_{3,i,G}, \ldots, u_{D,i,G}]$ [13]. For the DE algorithm, there are two different methods to perform the recombination process: the *exponential* (or *two-point module*) and the *binomial* (or *uniform*) [42]. In the exponential method, a random integer number n is chosen among the numbers $[1, D]$, where D indicates the number of chromosomes on the vector. This number will be used as a starting point in the target vector, from where the chromosome started its exchange with the donor vector. A second integer value L (from Lenght) is chosen from the same range $[1, D]$. This second number indicates the number of chromosomes that will be in fact read from the donor vector. Next, the CR (*Crossover Rate*), which is a control parameter for the DE just like the F, will be used to decide, whenever the resulting vector overwrites its current chromosomes with the values read from the donor vector.

The resulting vector will be evaluated and scored by the fitness functions and then compared with the current population vector $X_{i,G}$. If it presents a better score, it will replace the current vector on the population. If the current vector has a better score, then the new vector will be discarded. But when two or more fitness functions are used, it might not be possible to define which vector has the best score between them. When this happens, both vectors are kept in the population and only when the current iteration (generation) ends, will they be processed; the population will be trimmed to keep only the best individuals.

2.2.2 Simulated Annealing

In metallurgy, *annealing* is a heat treatment used to change the physical and chemical properties of a material in order to increase its ductility and reduce its hardness. The material is heated and then subjected to a slow cooling period. The Simulated Annealing algorithm (SA) was created in the context of the mechanical statistic [22] and was developed by Kirkpatrick, Gelatt and Vecchi in 1983 [28].

The SA is used to solve Combinatorial optimization problems, such as $min_x f(x), x \in S$, where $f : S \to \mathbb{R}$, with finite S. In this context, the optimization process is executed by levels, simulating the temperature levels during the cooling process. For each level, given a point $u \in S$, many neighbour points around u are generated, and the corresponding f value is calculated. Each generated point is accepted or refused according to a certain probability. The probability of acceptance decreases according to the process level, which means it depends on the temperature [22].

Algorithm 2.1 shows the SA basic steps. $T_k \in \mathbb{R}_+^*$ represents the temperature at level k and L_k the number of points generated at this level. Initially, T_0 and L_0 are fixed and an initial point u is chosen from S, according to [22].

Algorithm 2.1: Simulated Annealing pseudo algorithm.

begin
  Initialize u, T_0, L_0
  $k \leftarrow 0$
  repeat
    for $l \in [1, ..., L_k]$ **do**
      Generate w from $V(u)$
      if $f(w) \leq f(u)$ **then**
        $u \leftarrow w$
      else
        if $random[0, 1) < exp(\frac{f(u-f(w))}{T_k})$ **then**
          $u \leftarrow w$
        end
      end
    end
    $k \leftarrow k + 1$
    Calculate L_k and T_k
  until *'stop criterion'*
end

In order to skip local minimum values, the idea is to accept almost every proposed transaction. It is important to make sure T_0 is big enough, and then, use diminishing probability rates to accept the points that get the worst score values when evaluated by the fitness function. When the limit $T_k \to 0^+$ is reached, only the points getting the best scores from the fitness function will be accepted [22].

2.3 Task Mapping Benchmarks

Benchmarking is the quantitative foundation of computer architecture research. Without a program selection that provides a representative snapshot of the target application space, performance results can be misleading and no valid conclusions may be drawn from an experiment outcome [5]. Parallel architectures, such as *Chip-Multiprocessors* (CMPs), may unleash its full potential when running software that matches its hardware features. It means traditional sequential software may not be suitable to these architectures, in the same way that traditional benchmarks may not attend the requirements of modern hardware projects.

Future applications will have to be parallel; but due to the lack of a representative, and multi-threaded benchmark suites, most scientists are forced to fall back to existing benchmarks. This usually means the use of older *High-Performance Computing* (HPC) workloads, smaller suites with only a few programs or unparallelized benchmarks [5].

Some benchmark solutions were launched to tackle these shortcomings. The *Princeton Application Repository for Shared-Memory Computers* (PARSEC) was first launched by Intel Corporation and Princeton University in 2008 [4] and employed for benchmarking more than 55 papers in the International Symposium on Computer Architecture (ISCA) from 2010 to 2014 [50]. The PARSEC benchmark proposed to reach five goals:

- **Multi-threaded Applications:** multi-processor architectures are the most employed models nowadays for High-Performance Computing (HPC) systems. The trend for future architectures is to reach performance improvements by increasing the number of processing elements. Thus, the benchmark must support parallel applications.

- **Emerging Workloads:** the increasing of processing power comes attached to growing workloads. A new class of applications may require new capabilities placed beyond current architectures.

- **Diversity:** a benchmark must be representative of different application categories.

- **Employ State-of-Art Techniques:** a benchmark must be updated with the current state-of-the-art techniques.

- **Support Research:** different properties must be provided by a benchmark designed to attend not only commercial evaluations but also researching applications.

The PARSEC benchmark includes applications from various domains, such as financial analysis, computer vision, enterprise storage, data mining, etc.

2.3.1 NASA NAS Parallel Benchmark

The *Numerical Aerodynamic Simulation* (NAS) *Parallel Benchmarks* (NPB) is another benchmark for testing the capabilities of parallel computers. The main difference of this benchmark from PARSEC is its ability to select some parallelization techniques, and provide the shared memory and message-passing programming models freely. The two main frameworks are OpenMP – for shared memory – and MPI – for message passing – and some additional frameworks [16].

Applications from the NASA NAS Parallel Benchmark are derived from computational fluid dynamics, and are used for astronautical applications [47]. These applications are distinguished from traditional scientific applications due to the large memory requirements [56]. Although the NPB suite is grounded in problems of computational fluid dynamics, they are valuable in the evaluation of parallel computing since they are rigorous and as close to real applications as can be expected from a benchmark suite [2]. It makes this benchmark ideal for evaluating parallel applications running in a Network on Chip (NoC).

The NASA NAS Parallel Benchmark is composed of eleven applications, which mimic standard computation and data movements in CFD applications [16]. Five of the eleven applications were chosen to be used:

- **MG:** *Multi-Grid* on a sequence of meshes, long and short distance communication, and memory intensive. It is a kernel to solve the Three-dimensional Poisson equation using a simplified computing model. It is based on constant rates rather than on variable coefficients as do more realistic solvers. This test is ideal for long distance, highly structured communication based applications.

- **CG:** *Conjugate Gradient*, random memory access and communication. It is a method used to compute an approximation of the smallest eigenvalues of a large, sparse, symmetric positive definitive matrix. Complementary to MG, it tests irregular long distance communication, employing unstructured matrix-vector multiplication.

- **FT:** Discrete 3D Fast *Fourier Transform*, all-to-all communication. It is an alternative 3D partial differential equation solver using Fast Fourier Transform. It tests long communication performance.

- **IS:** *Integer Sort*, random memory access. It moves a huge amount of data to execute sorting operations. By contrast to other applications in the same benchmark suite, it uses integer arithmetic operations only, in a way floating-point operations are not involved.

- **LU:** *Lower-Upper* Gauss-Seidel solver. It solves a synthetic system of nonlinear partial differential equations using a symmetric successive over relaxation solver. LU is very

sensitive to the performance of the communication infrastructure. It sends a large number of short messages (40 bytes each).

These applications were selected because they have task communication based profiles, therefore they are ideal for the purposes of this work.

3. RELATED WORK

The subject of partitioning and mapping applications onto Network on Chip architectures have been exhaustingly explored [41]. This chapter analyzes previously documented work relating to task partitioning and/or the mapping problem, with a focus on mesh shaped NoCs. Works based on evolutionary approaches, specifically on the DE, were prioritized.

C. A. M. Marcon et al. [36] tackled the task mapping problem under the energy consumption and execution time views. To decrease energy consumption, application tasks are mapped to the target architecture considering the most efficient communication possible. This goal is reached by modelling computation and communication characteristics of each core. Different applications graphs may be used as an input, depending on the level of the computing model desired. The first model is the *Communication Weighted Graph* (CWG), representing the cores of an application and the messages exchanged between them. The second graph model is the *Communication Dependence and Computation Graph* (CDCG), which contains the messages exchanged between all tasks, the message sizes and also the dependence between them (the starting and ending points for each message to be sent). Thus, applications constituted by an arbitrary number of cores may be modelled by CDCG representing its computation and communication characteristics. Both models (CWG and CDCG) are evaluated by a deterministic XY routing algorithm. The third model used in this work is the *Communication Resource Graph* (CRG), which is a directed graph whose edges and vertices represent physical links and routers of the target architecture, respectively. This model is similar to a *NoC Topology Graph* [51] or an *Architecture Characterization Graph* [25]. The described models were used as starting points for task mapping efforts performed by two different algorithms: *Exhaustive Search* (ES) and *Simulated Annealing* (SA). NoC sizes ranged from 2x2 up to 12x10. For small NoC sizes (up to 3x4), both algorithms reached similar performance measurements. The authors claimed it was not possible to find optimum mapping solutions by using ES algorithm on bigger NoC sizes (from 8x8 to 12x10, for example) in a reasonable time. On average, CDCM based mappings tend to be 40% faster than CWM models in regards to execution times, whilst the energy consumption reduction seems to be insignificant (less than 1%).

J. R. Ku and S. G. Ku [27] proposed the use of an evolutionary algorithm to solve the task mapping on NoC problems, striving to minimize both the energy consumption and the link bandwidth requirements. Three random benchmark applications were generated for the tests at the same time, and one real world M-JPEG encoder was also used. They divided their implementation in two distinct phases. The first phase was called *Task Assignment problem* (CT-GA) where they tried to group tasks with high communication volume with each other, which they believed should minimize the computational energy by reducing

power consumption. For this phase, they used a classical implementation of the NSGA-II algorithm with an elitist approach. The second phase was called *Core-Tile Mapping* (CT-GA) and its goal was to associate the previously generated task groups to PE onto the NoC. For this phase they used a modified implementation of the NSGA-II algorithm with a particular implementation of the mutation genetic operation, where they tried to place most communicating task groups close to each other when possible. The authors claimed that their implementation found solutions 15% to 20% on average more efficient with energy consumption when comparing solutions generated by the *Physical Mapping Algorithm* (PMAP).

The work from *Isask'har Walter et al.* [57] analyzed the task mapping problem considering the steep increase in the number of transistors available on a chip and also the growing importance of keeping the energy consumption low. These issues lead to the existence of two trends: the availability of replicated modules that have identical functionality. The second trend is the creation of task-specific optimized modules (instead to general purpose PEs), used to prolong the path traversed by data on the chip. The authors argued the NoC design process should take into account this development, since the classic modelling patterns prevent exploiting the true benefits of a NoC based design. They have shown the importance of adapting the implementation of the task mapping problem to account classes of replicated modules and application-level latency requirements. They have tested this new formulation and claim the new approach leads to more efficient tile mapping and yield significant savings in communication costs. They used the *Cross-Entropy Method* (CEM) [40] to identify task sets and group them together. The CEM may be considered an evolutionary algorithm, but relies on statistical methods of sampling instead of genetic operators such as mutation or recombination. Random communication task graphs were generated for the tests. The authors claim they reached peaks of almost 35% savings when comparing their approach to a classical Simulated Annealing implementation.

S. Le Beux et al. [30] used an Evolutionary Algorithm to perform a multi-objetive optimization of the partitioning/mapping process. The inputs for this solution are the application model (constituted by a graph representing application tasks), computing loads (the number of cycles required for each task to run), the communication between tasks, and a graph representing the target architecture, which is normally a 2D mesh NoC. Three objective functions are used. The first objective is *system throughput*, which characterizes the system's ability to pipeline the application execution. The throughput corresponds to the number of application iterations the system can execute in a given period of time. The second objective function measures the *area cost*, indicating the area (space) necessary to implement a *System on Chip* (SoC). It summarizes the space of all cores allocated within the NoC on tasks from a specific application, resulting in a scalar value. The third objective is the *architecture flexibility*, which stands for the future ability to be updated to attend to new requirements. The idea is to minimize the number of allocated processors. Its measurement is given by dividing the number of tasks from the input application by the number

of allocated processors. The *Non-dominated Sorting Genetic Algorithm II* (NSGA-II) uses crossover and mutation genetic operations to evaluate the area cost and architecture flexibility of every candidate solution on each generation (iteration), while a simulation is performed to measure the throughput. The authors claim that partitioning and mapping decisions can be taken early because optimizations are performed at system level, which may contribute to reducing the time to market.

The work from *N. Nedjah et al*. [39] proposed a multi-objective evolutionary design tool to assist NoC designers at high-level stages of a platform based on the NoC design. It employed the *Communication Analysis For Embedded Systems* (CAFES) [37] and also used two multi-objective evolutionary algorithms: the *NSGA-II* and the *Micro Genetic Algorithm* (microGA). The application is represented by a directed acyclic task graph (TG) based on *Extensible Markup Language* (XML) technology, where each node on the graph represents a task and its execution and communication attributes. Objective functions are: *hardware area*, which stands for the silicon area needed to implement a NoC-based application; *execution time* and also *power consumption*. The maximum amount of tasks allocated to a single PE was set to 3. Non-valid mappings were kept on the population. Tests were run simulating applications from different domains, such as FFT, Inverse Fast Fourier Transform (IFFT), Pulse With Modulation (PWM) and others. Tests demonstrated that the performance of the NSGA-II was similar to that reached by CAFES, while the microGA appeared to have performed better than both, particularly in regards to the hardware area, where the microGA was almost 50% more efficient.

In [20] *D. Göhringer et al*. proposed a novel design methodology for helping architecture development and application partitioning on *Multi-Processor System on Chip* (MPSoC). The main contributions of this work were the tracing library, developed to generate timing information and a call graph from the sample application; a new function for partitioning the application source code in C++; a Profile Analyzer for helping identify hotspots within the code of local processors and which suggest more appropriated HW/SW Codesigns. The application is first subject to a partitioning phase, where tasks are grouped using the Hierarchical Clustering Algorithm. The partition groups are created in many steps, where the algorithm uses a closeness function at each step to decide which next two tasks will be added to an existing cluster, and then repeats this process until only one final cluster remains. This procedure creates a hierarchical list that indicates the ideal number of processors for the desired hardware architecture. The closeness function is responsible to guarantee equal workloads for every processor, minimize the overhead for message exchange between the processors, respect the memory constraints of each processor, and finally to reduce the power consumption by decreasing the area size of the FPGA. This work also uses a communication model grounded on an heuristic approach, by trying to identify better task associations by positioning message switching tasks as neighbours, when possible.

C. Deng et al. [14] adapted the classical DE to adhere to a high-level task model, free of a target architecture. A new permutation step was included right before the recombination phase of the classical DE, so that the current individual chromosomes would now be subject to a sorting procedure before selection. The authors claimed this modification was inspired on the *Ant Colony Algorithm* (ACA). Tests were made comparing the modified DE with the ACA, the *Improved Ant Colony Algorithm* (IAC) and the *Genetic Algorithm* (GA). The authors did not describe which fitness function was used for the tests, but they claimed the modified DE was superior to the ACA and GA, as a result of reaching the best fitness solution earlier (with a fewer number of computed generations).

The work from *E. Antunes et al*. [3] explored task-partitioning and processor-mapping methods on homogeneous NoC-Based MPSoC. The effects of both on the energy consumption of applications was explored separately and unitedly. Different application and architecture representations were used, such as the *Task Communication Graph* (TCG), representing application tasks and its exchanged messages; the *Communication Weighted Graph* (CWG), that is similar to the TCG format, however its vertices represent a set of processors instead of applications tasks; and the *Communication Resource Graph* (CRG), representing physical links and routers on the target architecture. A complete Energy Model was also created to represent the power consumption of processors, memory and communication architecture. Task partitioning and mapping were made in two different execution phases since the authors claimed direct task mapping solution might generate less efficient task distributions onto the target architecture. Results demonstrated that the mapping phase had a deeper impact on energy consumption, albeit the partitioning influence could not be neglected. Experiments were conducted on the CAFES framework.

Pavel Krömer et al.[29] presented an article describing a parallel DE implementation for GPU architectures based on the *Compute Unified Device Architecture* (CUDA) code. They have focused their code on solving the task scheduling problem on heterogeneous platforms. By following the *Single Instruction, Multiple Data* (SIMD), the computing model offered by a CUDA compatible GPU, they were able to create separated kernels (the basic program unit of a CUDA application) to perform each DE step: a first kernel to generate the initial population; a second kernel to evaluate population individuals; a different kernel to generate offspring individuals (covering mutation and recombination operations) and a final kernel to merge parent and offspring populations. The main advantage of this approach was to keep the processing data on the GPU memory during the entire execution time, saving precious time by preventing long data copies between host and accelerator memories. Since each thread was responsible for computing one individual concurrently, the population size was limited by the capacity of GPU running threads. The authors did not discuss the occupation rate of the GPU nor the quality of resulting solutions, but they claimed that the GPU implementation reached better solutions on two of three tests when compared with the sequential CPU version. No description has been made about how they solved the ran-

dom generation problem, since most GPU architecture simply do not support randomization routines.

In [23] *K. Hao, B. Wang and Y. Luo* proposed to solve the *Network Coding Problem* using a multi-objective optimization based in a modified NSGA-II implementation. Similar to the Task Mapping Problem, the Network Coding problem is also a NP-hard problem, which in turn, makes the evolutionary approach an interesting one to find reasonable solutions in feasible computing times. Two fitness functions were used to evaluate coding cost and link cost solutions. An elitist strategy was used in an attempt to keep the population diverse. The authors claimed the proposed algorithm computed acceptable solutions in a feasible time.

Sen Zhao et al. [60] proposed a Multi-Objective Differential Evolution algorithm based on adaptive mutation strategies and a partition select search to further improve diversity and convergence on generic multi-objective problems. The mutation operation was adapted to use the *DE/rand/1/bin* strategy for best ranked individuals in the population, while not so good ranked individuals were subject to a *DE/current to best/2/bin* strategy. Using this strategy to calculate differential vectors from individuals in the same frontier may favour the dispersion of the generated mutated individuals along that frontier and also help to spread the information to reach a non-dominated set with good distribution characteristics. Also, the selection operation of new mutated individuals (donor vectors) were changed in order to now take into account the comparison between the new individual against all other individuals in the population, and not only against its 'father'. This is accomplished by creating a temporary global population on each DE generation. The F (mutation factor) suffers a steady decline on its value on each generation, which tends to decrease the disturbance on the population (possibly decreasing its diversity) but still plays an important role in refining the local search, which may be beneficial to carrying it out and possibly accelerate the algorithm convergence. Tests were made using the five classical ZDT benchmark functions [62] comparing the proposed approach, the NSGA-II, a classical MODE and the *Strength Pareto Evolutionary Algorithm 2* (SPEA2). The benchmark compared the mean and variance metric convergence of all testes implementations. The authors claimed the proposed MODE variant presented an improved behaviour, while keeping its classical features of a faster convergence and good diversity. They also have affirmed that their novel implementation presented superior results in some cases.

The work by *S. Umamaheswari et al*. [54] proposed the *Firefly Algorithm* (FA) [18] to perform a multi-objective task mapping onto irregular NoCs, seeking for generate solutions which could yield a lesser power consumption and smaller areas allocated on the chip. The 'task scheduling' step may be associated to the classical partitioning problem, whilst the 'core mapping' corresponds to the classical task mapping problem from previous works. The NoC cores are associated as tiles. The task scheduling creates groups so that the summation of execution time of tasks within the group can not exceed the maxim value allowed for each tile on the NoC; the total number of tasks should be less or equal to the

maximum number of tasks allowed on the NoC tile and the total communication rate should be less than the total available bandwidth of the NoC surrounding links. Every NoC PE can allocate only one task at time. A comparison between the proposed FirelFly algorithm with the classical NSGA-II algorithm suggests the new approach seemed to generate optimal candidate solutions fasters the the NSGA-II. The authors claimed that their implementation was sensibly faster than the NSGA-II, specifically when computing bigger area NoCs. No comparison between the quality of generated solutions have been shown.

Mengyuan Wu et al. [58] suggested a new approach to reduce computing time on NoC simulations during design time. They proposed a new algorithm called *NoC Design Optimization based on Gaussian Process Model Assisted Differential Evolution* (NDPAD). This model used the DE algorithm to generate candidate solutions that were then subject to a second optimization phase where these individuals have their evaluation predicted by an individual solution-based training data selection method. This prediction was supported by a historic database of individuals. Only the most promising individuals were subject to a complete NoC simulation that indicated which ones were the best. Tests were performed using 6x6 NoC sizes in order to prevent extra long execution times, although the authors have said this scenario would favour traditional EA due to the smaller solution space to be searched. A comparison of their novel method was made against the *Selection-Based Differential Evolution Algorithm* (SBDE) [61], which is a modified version of the traditional DE. Results suggested the novel approach may be around 50% faster then the traditional methods when using a specific set of values for the *CR* and *F* execution parameters.

In [43], Z. Qingqi et al. tried to solve the task mapping on the NoC problem using a two objective evolutionary algorithm to reduce energy consumption and communication latency by optimizing the distribution of the link load. They have created an optimization method bonding the Membrane Computing and based on Mixed Structure and Genetic Algorithms (MCGA). Membrane Computing is a bio-inspired method used to find new computational models from the study of biological simple organisms such as living cells. This model also allows the use of a parallel approach to searching the solution area. Two applications were used as benchmarks, a *Video Objective Plane Decoder* (VOPD) and a MPEG-4 decoder. Tests indicated the novel approach was slightly more efficient than the traditional GA implementation, leading to solutions which were around 10% more economic in energy consumption, 35% faster in regards to communication latency for the VOPD benchmark, around 13% more economic in energy consumption and 3% faster in communication latency for the MPEG-4 application.

In [45], *A. Roy et al*. presented a hybrid technique associating the classical algorithms *Particle Swarm Optimization* (PSO) and the *Neighborhood Treemap* (NMAP) to reach better results on the task mapping onto the NoC problem. Benchmark applications were generated by the TGFF tool [15]. These applications covered different domains, such as multimedia applications (MPEG and MP3) and generic task graph implementations. These

applications ranged from 12 tasks up to 64 tasks, and mappings were made onto 2x4 NoCs up to 8x8 NoCs. Two modified versions of the hybrid NMAP + PSO implementations were proposed and compared to two classical algorithms, the NMAP, the *Integer Linear Programming* (ILP) and the *Kernighan-Lin* bi-partitioning strategy (KL) [46]. The authors commented that the best solutions were found by the ILP implementation, although this technique was not feasible on larger task graphs due to its computational cost. Results showed one of the modified implementations NMAP + PSO were able to reach the same optimum solutions as the ILP technique in some cases. When comparing the hybrid implementations to the KL technique, the proposed approaches found better solutions in six of eleven executions.

This work presented a DE Algorithm for combinatorial optimization *A. Maravilha et al*. [34] using a set-based representation and a special approach to explore the search space. The benchmark application was a *Capacitated Centered Clustering Problem* (CCCP) implementation. The proposed adaptation employed DE to define sub-problems covering a range of elements across candidate solutions. The authors claimed that the DE implementation was able to return competitive solutions when compared to a ILS implementation.

B. Xue, W. Fu and Mengjie Zhang [59] used modified versions of DE on single-objective and multi-objective modes as an approach for selecting a set of non-dominated feature subsets, which included a small number of features. The approach achieved high classification performance. Both DE implementations were evaluated after computing datasets from the UCI machine learning repository [19]. For each selected dataset, the instances were randomly separated into two distinct sets: 70% as the training set and the remaining 30% as the test set. The crossover rate was set to 0.3. Every test was performed for 30 independent runs on each dataset. This was important, since DE is a stochastic algorithm. Results have shown that both, *Multi-Objective Differential Evolution* (MODE) and *Single-Objective Differential Evolution* (SODE) implementations outperformed traditional *Feature Select Algorithms* (LFS).

The work from *D. Das, M. Verma and A. Das* [12] tackled the hardware-software partitioning problem using a DE implementation. The goal was to design the application tasks distributed into specific hardware and software parts to adhere to real time constraints as well to reduce system cost. Therefore, the real challenge depended on choosing which tasks should be implemented in hardware and which ones in software. This is a NP-Hard class problem. To a certain extend, a brute force or deterministic algorithm would not be feasible due the large running time it would require. Their implementation used values 0 (for hardware allocation) and 1 (for software allocation) for chromosome representation. It meant a data structure representing an individual on the population would be as long as the number of existing tasks to be distributed. The fitness function considered the following quantities: execution time, area cost, communication cost between software and hardware partitions. Tests were executed using sets sizes ranging from 10 up to 30 tasks. A comparison was made with a *Particle Swarm Optimization* (PSO)algorithm, both implementations

(DE and PSO) have been coded for running in a MathLab environment. The authors have not discussed the quality of resulting solutions, but they claimed that DE presented a smaller execution time, reaching the termination condition (number of generations or a predefined fitness value) faster than PSO. Results have shown DE is around 46% faster than PSO for smaller task sets (10 tasks) and 37% faster on average for bigger task sets (30 tasks).

O. Cortes et al. [10] proposed an exploration on the influence of execution parameters on speeding up DE when running on *Graphic Processor Units* (GPUs). Although this work focused on high level DE implementations, test results suggested the dimension size (number of chromosomes on each individual) and the number of calls to the evaluation functions may have deeply impacted the execution time and also affected the quality of resulting solutions. Base tests ran solely on CPU have shown the quality of the solutions tends to increase when using lower level values for the CR. On GPU, this behaviour does not seem to occur, although the average quality of the solutions seems did not seem to reach the same quality obtained from CPU executions. The main contribution of this work is the claim that a very specific parameter setting should be chosen for a specific problem being tackled with a DE implementation.

In [1], *A. Al-Wattar et al*. proposed a new framework using an *Island Based Genetic Algorithm* (IBGA) flow that optimized several objectives including delay and power consumption. This method made use of parallel and isolated populations from different locations on the solutions space. Each population was processed by a particular GA process. Best candidate solutions from each population were then evaluated and generated separated Pareto fronts. Only the best ranked solutions were finally extracted and globally compared. Besides allowing a faster exploration of the solution space, this method also has the advantage of not only indicating an optimal performance and power profile, but hardware platforms (floorplans) may also be used as tests. This is an important feature of the dynamic reconfigurable systems. Tests were executed using applications from different domains, such as JPEG, MPEG, EPIC, MESA and HAL benchmarks. When comparing these results with the conventional GA, the proposed IBGA implementation achieved on average 55.2% improvement over the single GA implementation.

Table 3.1 shows a summarized comparative of the works described in this section.

Table 3.1: Summary of Related Work

Work	Year	Application	Target Architecture	Algorithm	Goal
[36]	2005	task mapping	generic NoC	Exhaustive Search and Simulated Annealing	reduce energy consumption and communication cost
[27]	2007	task mapping	generic NoC	modified NSGA-II	minimize energy consumption
[57]	2009	task mapping	generic NoC	Cross-Entropy Method	reduce communication cost
[30]	2010	NoC design	generic NoC	NSGA-II	reduce time to market
[39]	2010	NoC design	generic NoC	NSGA-II, microGA and CAFES	support NoC design
[20]	2010	generic	generic	Hierarchical Clustering	support software and hardware design
[14]	2010	non-specified	high-level abstraction	Differential Evolution, Ant Colony and Improved Ant Colony Algorithm	algorithm comparison
[3]	2011	task partitioning and task mapping	generic NoC	CAFES	explore the impacts of task partitioning and task mapping on energy consumption
[29]	2011	task scheduling	SIMD Graphic Processor Units	parallel DE	reduce execution time
[23]	2012	network coding	generic network	modified NSGA-II	solve the network coding problem
[60]	2013	ZDT benchmark functions	non-applicable	modified DE	explore a modified approach for individual selection and a dynamic adaptive mutation strategy
[54]	2013	NoC design	generic NoC	Firefly and NSGA-II	silicon area and energy consumption optimization
[58]	2014	NoC design	generic NoC	Differential Evolution	reduce computational cost of NoC simulations during design time
[43]	2014	task mapping	generic NoC	modified GA	reduce energy consumption and communication latency
[34]	2014	Capacitated Centered Clustering Problem	non-applicable	modified DE	explore DE capacities with altered genetic operators
[59]	2014	selection algorithm	non-applicable	Differential Evolution	explore SODE and MODE as a dataset selector for machine learning support
[12]	2014	hardware-software partitioning	generic	Differential Evolution	reduce execution time and communication cost
[10]	2015	CPU and GPU benchmark	high-level abstraction	Differential Evolution	optimization of execution parameters
[45]	2015	task mapping	generic NoC	ILP, KL, NMAP, PSO and hybrid NMAP + PSO	reduce communication cost
[1]	2016	map execution units to task graphs	generic FPGA	Island Based Genetic Algorithm	reduce energy consumption for static and partial dynamic reconfigurable systems
this work	2016	task mapping	generic NoC	Differential Evolution	reduce communication volume

4. PROJECT METHODOLOGY

This chapter describes the methodology used on this work, including the two Differential Evolution implementations created to solve the Task Mapping on NoC problem.

4.1 Modelling of the Data Structures

There are many different alternatives to define a data structure that represents a task mapping candidate solution. Basically, it is necessary to precisely indicate the NoC's location where each task is allocated. Ideally, we wanted to create a data structure that was possible to be used on both computing models we were working on, the first one that accepts one or more tasks allocated per PE, and the second model that allows only one task per PE.

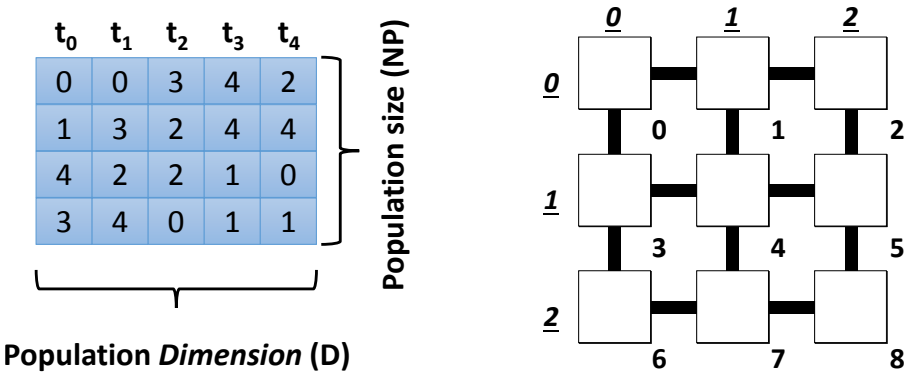


Figure 4.1: Population data structure.

The structure is represented in Figure 4.1 as follows: considering that the target architecture is a 2D mesh, a PE's location is given by $P = (x * w) + y$, where P is the PE's location, x indicates the line where the PE is positioned within the mesh, y indicates the PE's column and w indicates the NoC width (number of existing columns). In the given example, tasks t_0 and t_1 from the first individual in the Population were allocated to PE(0, which is located at the first line and the first column within the NoC. At the same time, task t_3 was allocated to the PE(4), which is positioned right in the middle of the NoC at line 1, column 1.

The number of columns is dependant on the number of chromosomes, which corresponds to the number os tasks from the input application.

4.2 Multi-Objective Differential Evolution Approach

The first explored application model is formed by a generic 2D NoC mesh where one or more tasks can be allocated simultaneously onto the same PE. Each PE can execute multiple tasks at a time and its local memory can store data from one or more tasks at the same time. Thus, tasks allocated within a PE always will be executed in parallel. Therefore, in this model, there are two parallelism levels: one or more tasks running simultaneously within the same PE, and also multiple PEs running concurrently. One important aspect of this model is the fact that tasks within a PE must be prioritized somehow. However, since optimization of execution time is out of the scope of this work, our model will not implement sorting nor prioritization among tasks allocated within a PE. Instead, two objectives were focused: the *Communication Volume Metric* and the *Load Balance Metric*.

4.2.1 Communication Volume Metric

For evaluating the communication volume of a task mapping candidate solution, this model considers the summation of all sent and received messages by tasks allocated within a PE. If two communicating tasks are allocated within the same PE, then their messages will not contribute to the total amount of communication, since their messages will never reach the NoC infrastructure of communication.

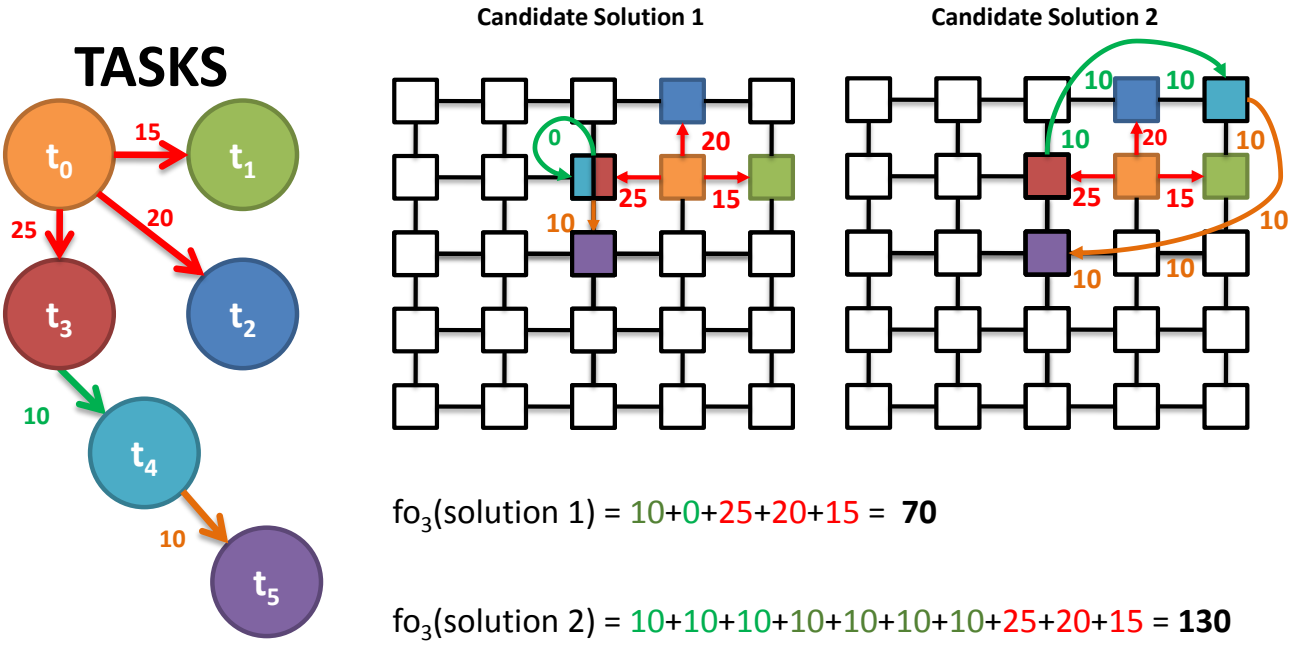


Figure 4.2: Evaluating the communication volume of two task mapping candidate solutions.

Figure 4.2 shows examples of two distinct task mapping candidate solutions and how they will be evaluated according the communication volume metric. The task set shows

messages sent from one task to another, including the message size (the indication of a specific measuring unit for the message size here is not significant). For example, task t_0 sends messages with different sizes for t_1 (15), t_2 (20) and t_3 (25); t_3 sends a 10 sized message to t_4, which, in turn, sends a 10 sized message to t_5.

The candidate solution 1 have allocated tasks t_3 and t_4 onto the same PE. Since the message sent from t_3 to t_4 will never consume any resource from the NoC's communication infrastructure, this message will not collaborate to the communication volume metric, as shown in Figure 4.2. On the other hand, on the candidate solution 2, these same tasks were allocated at different locations onto the NoC, in a way the message sent from t_3 to t_4 has now to cross a 3 hops long distance to reach its target. It means this message will access the send/receive engine three times until its sending process to be completed. For this reason, its contribution for the communication volume metric will be equal to 10+10+10, representing one summation for each dispensed hop. The number of hops used for this measurement will be calculated by using the *Manhattan Distance* metric, which is the distance between two points measured along axes at right angles. Considering a plane containing a point p_1 at (x_1, y_1) and a second point p_2 at (x_2, y_2), the Manhattan Distance (M_d) is given by the Equation 4.1.

$$M_d = |x_1 - x_2| + |y_1 - y_2| \tag{4.1}$$

For the given example, considering that the objective is reduce the total communication volume, the candidate solution 1 will be considered more efficient than the candidate solution 2, due the fact it returns a smaller value for the communication volume metric (70 for the candidate solution 1 and 130 for the candidate solution 2).

4.2.2 Load Balance Metric

The Load Balance Metric measures the variance of the processing load of all PEs on the NoC. It means that different PEs allocating tasks with different amount of CPU load will generate unbalanced candidate solutions. This metric has the goal to foster the creation of more balanced candidate solutions, by distributing more equally possible the processing load among NOC PEs.

Each task uses a particular amount of the CPU processing power, indicated as shown by Figure 4.3. For the given example the respective task's CPU loads are: t_0 (75%), t_1 (53%), t_2 (50%), t_3 (75%), t_4 (10%) and t_5 (65%). For example, it means t_0 will occupy 75% of the total computing time of the PE where it is allocated.

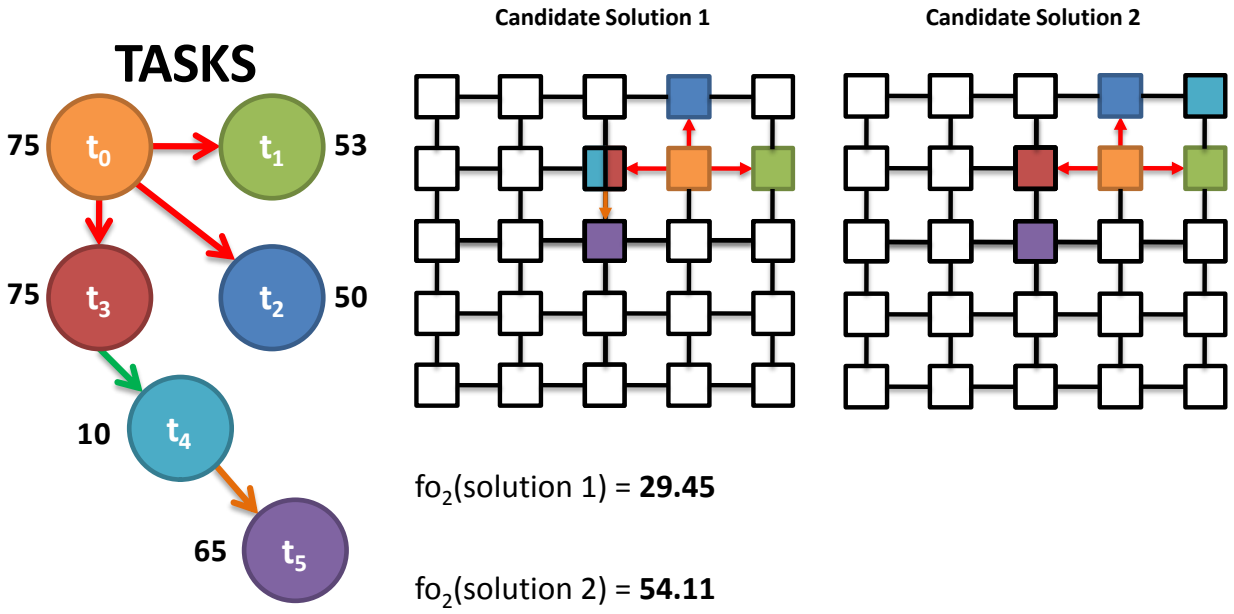


Figure 4.3: Evaluating the Load Balance metric of two candidate solutions.

The metric is calculated as follows. First, the summation of the CPU loads from all allocated tasks is computed for each PE using the Equation 4.2, where *i* identifies the PE; *k* is the number os tasks allocated at the PE *i*; and *L* is the CPU load of the task *j*.

$$\mathcal{X}_i = \frac{1}{k} \sum_{j=1}^{k} L_j \tag{4.2}$$

Next, the average value ($\overline{\mathcal{X}}$) is calculated by summing all total CPU loads of all PEs (PEs with no allocated tasks are ignored on this calculation), as shown in Equation 4.3.

$$\overline{\mathcal{X}} = \frac{1}{n} \sum_{i=1}^{n} \mathcal{X}_i \tag{4.3}$$

Since the computation of the average ($\overline{\mathcal{X}}$) is done, the variance (ϕ) is finally calculated as shown in Equation 4.4. Again, PEs with no allocated tasks will be ignored on this calculation.

$$\phi = \frac{1}{n} \sum_{i=1}^{n} (\mathcal{X}_i - \overline{\mathcal{X}})^2 \tag{4.4}$$

Finally, the value of the *Root-Mean-Square Deviation* (RMSD) of the candidate solution is calculated as shown in Equation 4.5.

$$RMSD = \sqrt{\phi} \tag{4.5}$$

A *RMSD* value closer to zero identifies a better CPU load throughput in the NoC PEs.

4.2.3 The Multi-Objective Differential Evolution Applied to the Task Mapping onto NoC problem

This section describes the adaptation of the DE to solve the task mapping onto NoC problem considering two optimization constraints: achieve a better distribution of task allocation over the NoC, by equalizing the processing load of the NoC PEs, and reduce the total communication volume. Because two fitness functions are used to evaluate the candidate solutions, this implementation is considered a *Multi-Objective Differential Evolution* (MODE) algorithm.

A notation to represent this model is in Equation 4.6. Being i the current individual in the population, G is the current Generation, D is the population *dimension* (the number of existing chromosomes on each individual), and NP is the population size (the number of existing individuals in the population).

$$x_{i,G} = [x_{1,i,G}, x_{1,i,G}, \dots, x_{D,i,G},] \quad i = 1, 2, \dots, NP. \tag{4.6}$$

To represent tasks allocation locations within the NoC, we have modelled the Population data structure as follows:

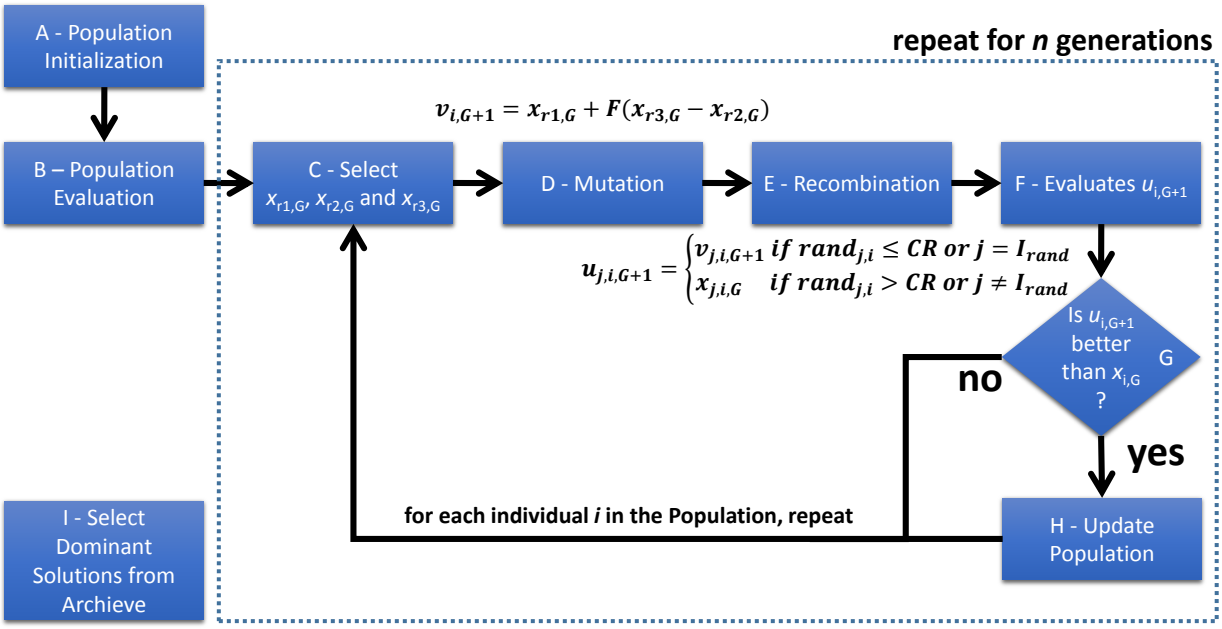


Figure 4.4: Multi-Objective Differential Evolution algorithm base procedures.

Base procedures of the MODE are shown in Figure 4.4. The basic routines are described as follows:

A - Population Initialization: first the lower and upper bounds must be defined, as in Equation 4.7.

$$x_j^L, \leq x_{j,i,1} \leq x_j^U, \tag{4.7}$$

For the underlying model, each chromosome value is randomly selected from the uniformly distributed interval $[x_j^L, x_j^U]$. L and U are execution parameters set by the user and must reflect the number of existing PEs on the target architecture. For example, a 3x3 mesh NoC will index its PEs from 0 (L) to 8 (U).

Next, the population is randomly initialized with NP individuals compound by D chromosomes each. Each one of the NP individuals undergoes mutation, recombination and selection procedures.

B - Population Evaluation: all individuals from the population are subject for evaluation from the two fitness functions f_1 and f_2.

C - Selection: for a given individual $x_{i,G}$, three vectors $x_{r1,G}$, $x_{r2,G}$ and $x_{r3,G}$ are randomly selected from the population, providing that the indices i, $r1$, $r2$ and $r3$ are distinct.

D - Mutation: the goal of the mutation procedure is to expand the search space. For that, a donor vector ($v_{i,G+1}$) is created by adding the weighted difference of two of the selected vectors to a third one, using on this calculation the mutation factor (F), as shown in Equation 4.8. For this model, F is a constant execution parameter set for a value from $[0.0, 1.0]$.

$$v_{i,G+1} = x_{r1,G} + F(x_{r3,G} - x_{r2,G}) \tag{4.8}$$

E - Recombination: the objective of the Recombination procedure is to assimilate good candidate solutions from previous generations. A new trial vector ($u_{i,G+1}$) is created from the chromosomes of the target vector $x_{i,G}$ (the current individual in the population) and the chromosomes of the donor vector $v_{i,G+1}$. Chromosomes from the donor vector will be included on the trial vector with CR probability, as shown in Equation 4.9, providing $i = 1, 2, \ldots, NP$ and $j = 1, 2, \ldots, D$.

$$u_{j,i,G+1} = \begin{cases} v_{j,i,G+1} & \text{if } rand_{j,i} \leq CR \text{ or } j = I_{rand} \\ x_{j,i,G} & \text{if } rand_{j,i} > CR \text{ or } j \neq I_{rand} \end{cases} \tag{4.9}$$

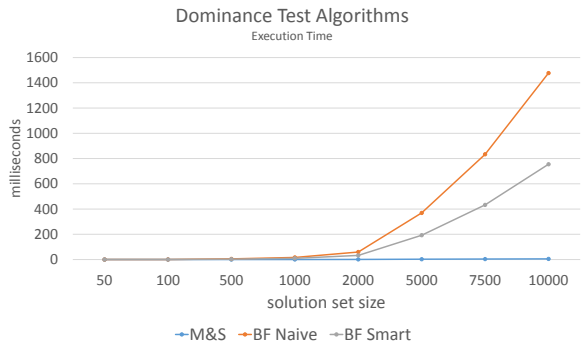
The CR is an execution parameter set by the user and must be ranged between $[0.0, 1.0]$. The parameter I_{rand} is an integer from $[0, 1, \ldots, D-1]$ (representing one of the existing chromosomes) and it is mandatory to provide that $v_{i,G+1} \neq x_{i,G}$.

F - Individual Evaluation: the new individual $u_{i,G+1}$ is subject to the two fitness functions f_1 and f_2 to compute its fitness values.

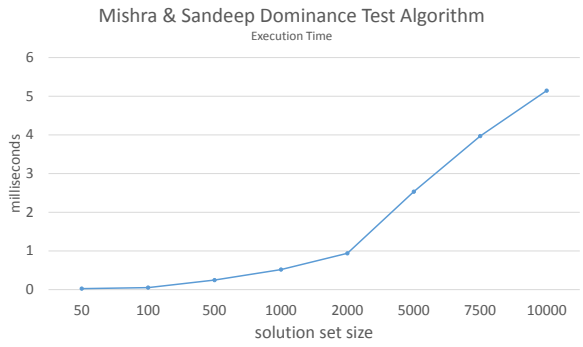
G - Dominance Test: the new individual (trial vector) $u_{i,G+1}$ is compared to the current individual (target vector) $x_{i,G}$ to identify who between them is dominant. This procedure uses the two fitness values from each individual to create a Pareto Front where is possible to find the dominant candidate solutions. According to [38],

> "...a solution x_1 is sad to dominate the other solution x_2, if both conditions 1 and 2 are true:
>
> 1. The solution x_1 is no worse than x_2 in all objectives, or $f_j(x_1) > f_j(x_2)$ for all $j = 1, 2, \ldots, M$.
> 2. The solution x_1 is strict better than x_2 in at least one objective, or $f_j(x_1) < f_j(x_2)$ for at least one $j \in \{1, 2, \ldots, M\}$."

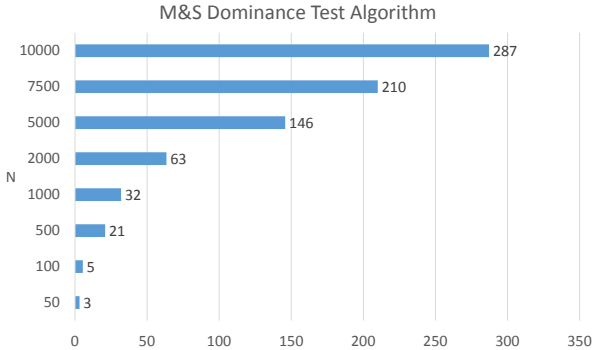
An ordinary brute force implementation of the dominance test would require $O(n^2)$ iterations to identify the dominant solutions (because it demands that each solution must be compared to all others). Since it represents a significant computing effort, we decided to test alternative dominance test algorithms to try enhancing our own implementation. We tested three different code implementations:



(a) comparison of execution times.



(b) M&S execution time detailed.



(c) speedup of the M&S algorithm.

Figure 4.5: Comparison between Dominance Test algorithms.

- **Brute Force "naïve"**: this is the conventional implementation with two nested looping structures and presents a complexity of $O(n^2)$.
- **Brute Force "smart"**: by creating a dependency in the counting control variable of the inner loop, it is possible to reduce the number of iterations by almost half, therefore the complexity falls to $O(\frac{n^2}{2})$, although it is still subject to a quadratic domain.
- **Mishra & Sandeep algorithm**: the M&S dominance test [38] uses an approach where every time a solution is found to be dominated by some other solution, it is removed from the solution set. Thus, at each new iteration the number of executed comparisons decreases, which prevents a large amount of computing effort to be executed. In fact, the complexity of the M&S dominance test is given by $O(N.log(N))$.

We tested these three dominance test algorithms by computing solutions sets ranging from 50 up to 10000 randomly generated individuals. The results are shown in Figure 4.5. In Figure 4.5a it is possible to see that the M&S algorithm presents a superior performance when compared to the other brute force implementations. This behaviour is specially perceived when computing big solution sets, such as 2000 or bigger ones. Since the difference of the M&S algorithm performance in relation to the brute force methods is so big, it is hard to note how this algorithm behaves when computing growing size solution sets using the same graph scale for both plots. Figure 4.5b shows a detailed graph displaying solely the M&S algorithm execution time. Even for bigger solution sets, its execution time remains below 10ms.

Figure 4.5c shows the speedup calculated for the M&S algorithm in relation to the Brute Force "naïve" implementation. The Brute Force "smart" implementation reached a top 1.9 speedup ratio for bigger solution sets, therefore it was not plotted on this graph.

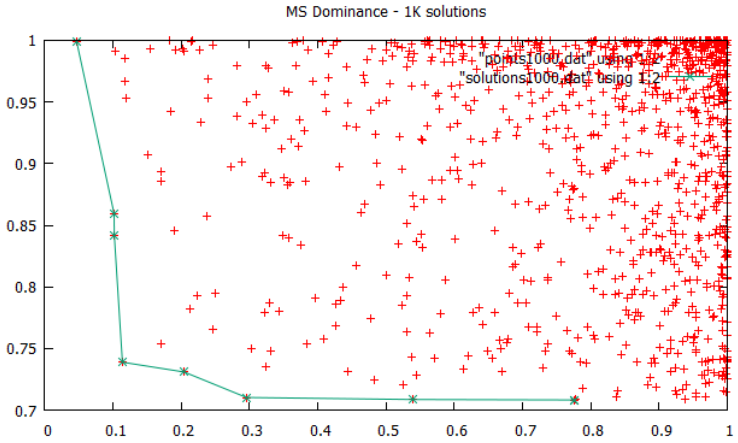


Figure 4.6: Example of Dominance Test executed with 1000 candidate solutions.

Since it was demonstrated there is no substantial difference on the execution time among these three tested algorithms for computing small solutions sets, and considering

our populations (and by consequence, the respective archive sets) would never be bigger than 30 individuals, we decided to implement the dominance test using lambda expressions supporting API[1]. This decision allowed to slightly reduce the complexity of our code, not only for the dominance test, but also for all archiving operations.

Finally, Figure 4.6 shows the results of a dominance test conducted using 1000 randomly generated candidate solutions. Dominant candidate solutions were highlighted by a green line forming a *Pareto Front*, which may not appear too round due to the unintentional distortion added to the vertical axis by the plotting utility[2] when this image was generated.

H - Update Population: this procedure defines which individuals must be kept in the population. The policy is defined by the Algorithm 4.1.

Algorithm 4.1: Update Population policy.

begin
  if $x_{i,G}$ *dominates* $u_{i,G+1}$ **then**
    keeps $x_{i,G}$ in the Population
    discards $u_{i,G+1}$
  end
  if $u_{i,G+1}$ *dominates* $x_{i,G}$ **then**
    adds $u_{i,G+1}$ to the Population
    removes $x_{i,G}$ from the Population and discards it
  end
  if *not* $((x_{i,G}$ *dominates* $u_{i,G+1})$ *or* $(u_{i,G+1}$ *dominates* $x_{i,G}))$ **then**
    keeps $x_{i,G}$ in the Population
    adds $u_{i,G+1}$ to the Population
  end
end

At the end of the current generation's main loop, it may be possible to see that the population size has exceed the NP size. When this happens, the Population must be trimmed back to the NP size before starting to compute a new generation. Dominant individuals must be prioritized, then followed by individuals with the best fitness values. At the same time, all dominant individuals are included into a temporary Population-like data structure called archive.

I - Select Dominant Solutions From Archive: the stop criteria for the MODE algorithm is the number of generations *n*. After computing all desired generations, the best (dominant) solutions found at each generation were appended to the archive data structure. As a result, it is possible that it now contains:

- repeated or equivalent candidate solutions;
- non-dominant candidate solutions.

[1] Introduction to LINQ Queries (C#) https://msdn.microsoft.com/en-us/library/bb397906.aspx
[2] Gnuplot - http://www.gnuplot.info/

For this reason, the archive content must first have its duplicates removed and then be subjected to a new dominance test, using the same *Dominance Test* procedure as described previously. The resulting individuals represent the best possible candidate solutions found by the MODE.

4.2.4 The Multi-Objective Task Partitioner Tool

A software tool was created to implement the described MODE model. Figure 4.7 shows a screenshot of the main window. This tool was coded using the Microsoft C# language for Windows Desktop environments including the LINQ framework, which is a library to support lambda notation-based queries.

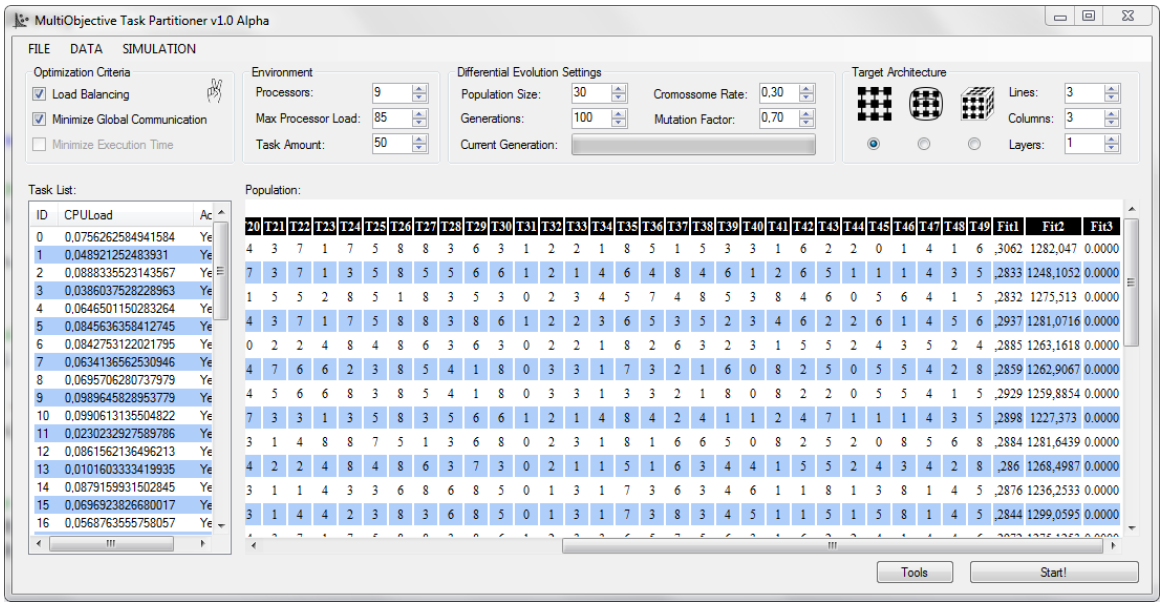


Figure 4.7: screen-shot of the MO Task Partitioner tool.

Implemented features are:

- **Task Generator**: it generates random tasks, where it is possible to set the number of desired tasks and the maximum CPU load (ranging from 1% to 100%). In this case, task's CPU loads will be randomly generated within the set range.
- **Support to CAFES task set format**: instead of generating random tasks, it is possible to load a task set from a CAFES formatted file. This featured was used to compare the results of task mapping tests we ran on the CAFES framework with results obtained from our tools. Both executions used the exact same initial data conditions.
- **Execution Parameters**: it is possible to configure the following parameters: the number of tasks (for randomly generated tasks only), the maximum CPU load range, the

number of PEs (generating 2D squared *NxN* meshes only), the Population size, the number of generations, the Crossover Rate (*CR*), and the mutation factor (*F*). With the MODE mode, the two objective functions are always used; although selecting only one will start the SODE mode (to be reviewed later in this document). It is also possible to use different target architecture configurations, for example, non-squared 2D (*NxM*) meshes, a torus mesh or even a 3D mesh. However, these extra models are outside the scope of this project and therefore will not be described.

When execution starts it is possible to see a progress bar increasing its length while advancing with the computation of the generations. Depending on the number of tasks and on the NoC size, the processing time may take a few minutes. In an Intel I5 1.7Ghz running Microsoft Windows 8 Professional, a sample application with 149 randomly generated tasks to be mapped onto a 9*x*9 mesh will take 6 minutes to complete on average.

At the end of the execution, several files will be generated. One Population file per generation is produced, containing all existing individuals in the Population at the end of each generation; one dominate set file, containing all dominant individuals at the end of each generation; an archive file, containing the final archive version at the end of the execution; and a log file containing the best and the worst fitness values of each generation.

In order to certify the MODE is converging properly and producing better solutions each time, we have used the Hyper-Volume metric. According to [31], the Hyper-Volume metric measures the area covered by all non-dominated solutions from a reference point, as shown in Figure 4.8.

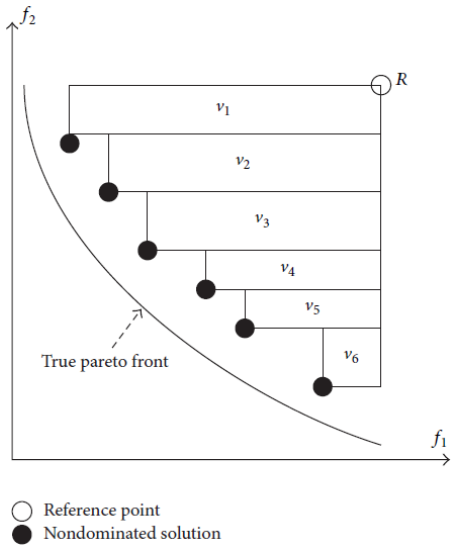


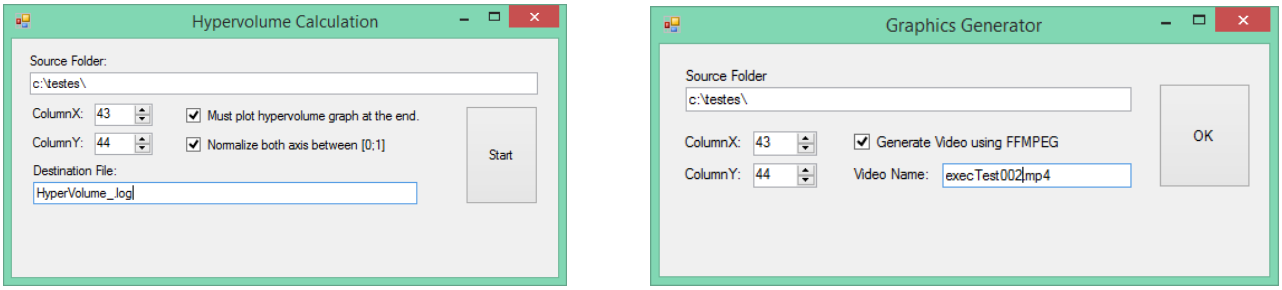
Figure 4.8: schema for calculation fo the Hyper-Volume metric. By [31].

We used the point $R = (1, 1)$ as a reference for this calculation. However, this is only valid if all values from both fitnesses were previously scaled to a [0.0 to 1.0] range. Thus, in order to avoid the introduction of any undesired biases, this calculation is performed only

after all generations have been computed. This will allow the identification of global minimum and maximum values which will eventually be used to reach the desired fitness of candidate solution from any of the processed generations. In order for a MODE implementation to run properly, it is expected that the Hyper-Volume metric will display a constant growth, or at least, never decrease (when optimizing minimization-based problems, like our model). This behaviour happens when a Pareto Front forms; the front will be positioned further away from the reference point each time. If the Hyper-Volume area suddenly had its area decreased, it would mean that a good candidate solution, that was previously close to the Pareto Front, was lost. Therefore, this metric is a safe indication of convergence used by Pareto Front-based Evolutionary Algorithms [31].

Tool set

A tool set was implemented to help running tests and also to validate results. The first is the Hyper-Volume Calculation tool, shown in Figure 4.9.



(a) Hyper-Volume calculation tool.

(b) Graphics and video generator.

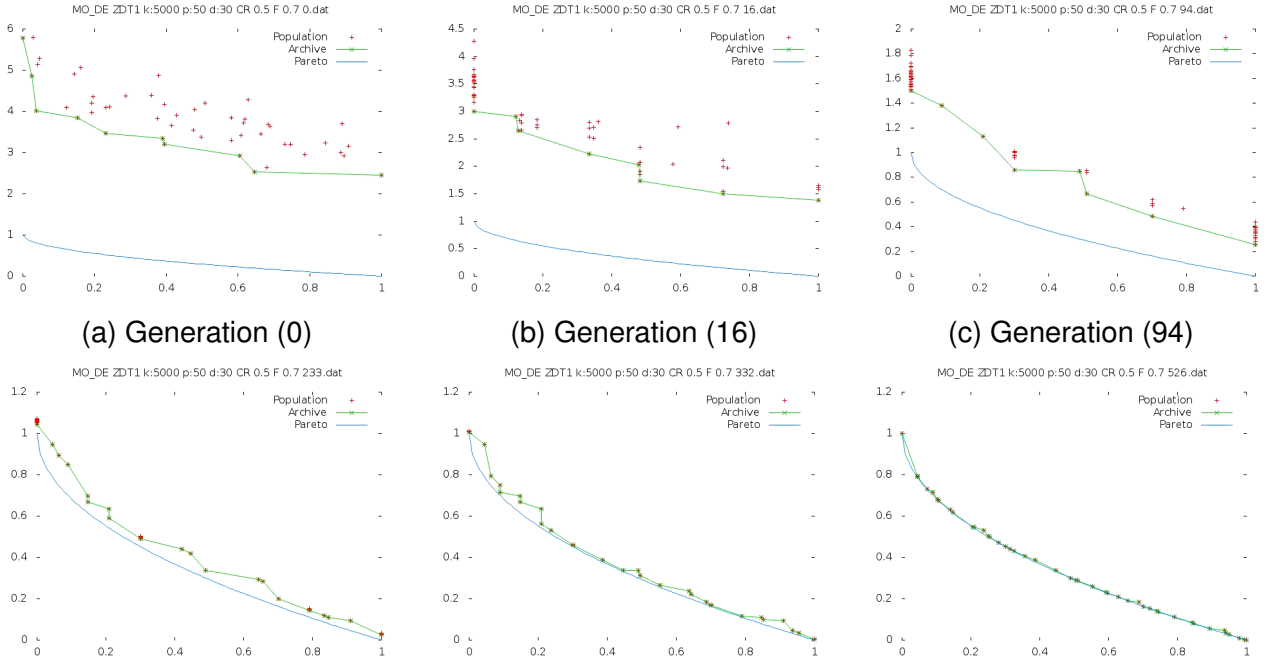
Figure 4.9: Tool set.

The Hyper-Volume Calculation tool, as shown in Figure 4.9a, reads output files generated by the last execution of the source folder and generates a log file containing the value of the hyper-volume which was calculated for each generation. Optionally, it can also plot this log file, and also scale global values to the $[0.0, 1.0]$ range.

The graphic and video generator tool is shown in Figure 4.9b. As in the same way the previous tool operates, it reads output files generated on the source folder and plots the Population, as well as the Archive (Pareto Front) obtained after computing each generation. Optionally, it can assemble an MP4 video to dynamically show how the MODE has behaved.

Figure 4.10 shows a benchmark application we have processed to make sure our MODE implementation was running properly.

In Figure 4.10a it is possible to check the generation 0, where an incipient Pareto Front starts to form, although still out of the range of the ZDT1 function. The Pareto Front is known to be subjected to the domain within the $[0, 1]$ period on both axis. During the next generations, as shown in Figure 4.10b and Figure 4.10c, this initial Pareto Front gets closer to the real one (shown as a blue curve). During the next generations, most solution points



(a) Generation (0) (b) Generation (16) (c) Generation (94)

(d) Generation (233) (e) Generation (332) (f) Generation (526)

Figure 4.10: Sample with the ZDT1 benchmark function.

will be placed right over the real Pareto Front. The speed of this convergence depends on the values chosen for the execution parameters *CR*, *F* and *NP*.

4.3 Single-Objective Differential Evolution Approach

The *Single-Objective Differential Evolution* algorithm (SODE) is useful for optimization problem that relies at only one constraint to be reduced (or increased). The second application model we explored was also formed by a generic 2D NoC mesh, but at this time, each PE can allocate only one task at time. Therefore, on this model, parallelism is achieved by running different tasks on different PEs simultaneously. Although this model focuses on reducing communication volume, the two fitness functions created for the MODE model can be subject to optimization individually, one at time.

The Communication Volume Metric works exactly the same way as it was coded to adhere to the MODE model. The difference relies on the fact that under this SODE, a candidate solution would never allocate more than one task onto the same PE at the same time. Thus, it was necessary to personalize the *Mutation* and *Recombination* procedures to respect this constraint.

Considering the SODE algorithm optimizes only one objective at time, this model requires lesser computing efforts in order to run. There is not the concept of dominance, given the fact there a Pareto Front is not feasible to raise. Instead, to check the algorithm

convergence it is necessary to follow the behaviour of fitness values among individuals in the population. For example, when the optimization problem seeks reducing a variable, a decreasing tendency must be noted. On the other hand, when the problem demands to magnify a variable, an increasing tendency must be registered among in individuals in the Population.

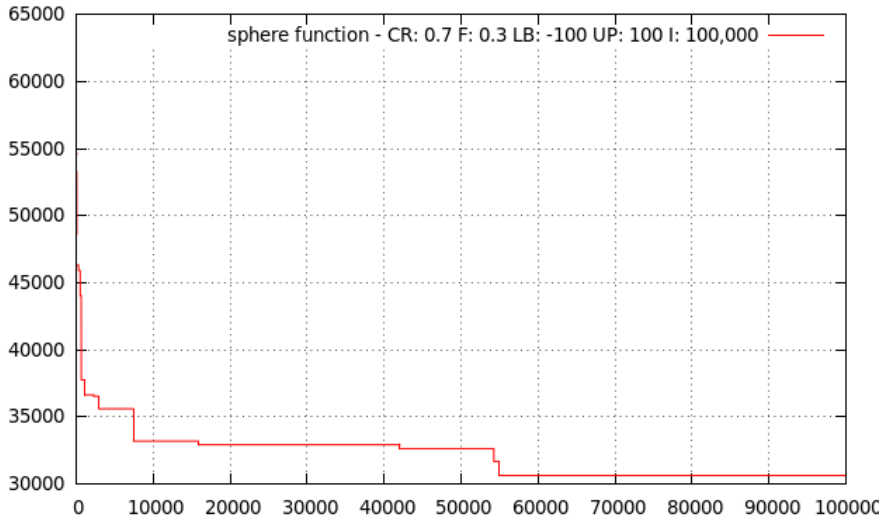


Figure 4.11: Single-Objective Differential Evolution running an Sphere function benchmark.

Figure 4.11 shows the results of our implementation of the SODE algorithm running an Sphere function benchmark. The value of the best solution in the Population (in this case, the individual with the smallest fitness value) is recorded into a log file and them plotted on this graph. Additionally, it is possible to record the value of the worst fitness from each generation set, and then create a more detailed view on how the algorithm is enhancing (or not) its Population.

4.3.1 Proposed Modification

For the proposed SODE approach, we have applied a modification inside the Recombination Genetic Operator in order to try to reward individuals that eventually may contain characteristic considered desirable to generate improved off-springs. This is achieved by using a pattern previously identified from the original task communication graph used as an input parameter. The method consists in trying to find which tasks are most communicating among them all, in a way that tasks that send and receive a bigger amount of messages will be considered as a core. Thus, all tasks sending or receiving messages from this core task should be kept at one hop distant from each other, including the core task. Figure 4.12 shows an example of this technique. First, a list containing all tasks that send messages are created. After that, this list is enhanced by adding on the same position tasks both tasks,

who send and receive messages from each other. At this point, a task might appear at more than one position at time. Finally, the message sizes are summed at each position on the list. The position containing the greatest value indicates the group of tasks that must be kept closer preferably. If two or more task group reach the same value, then only the first group met will be considered as a seed.

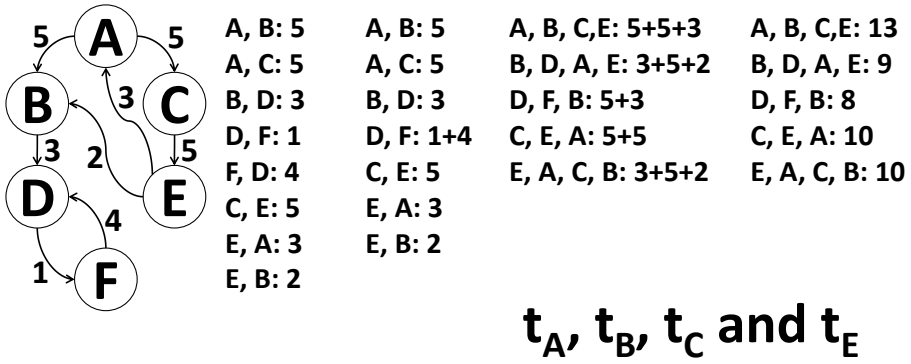


Figure 4.12: Identifying most communicating tasks.

5. EXPERIMENTAL RESULTS

This chapter presents and discuses the results of the experiments that were run using our SODE implementation to work with the Task Mapping onto NoC problem. The first section describes how the NASA NAS benchmark was used to evaluate our proposed implementation of the SODE. The next section further describes one of the tested benchmarks, the CG application. Following this section, the results of the test are discussed. Later, the next section describes a comparison between the results obtained by SODE and the CAFES framework. The last section presents some discussions.

A 'robot agent' was created to automatize the tests and to guarantee a smoother and more effective test execution. This robot reads an input file where each line represents a test case and informs the required execution parameters to run the test. These parameters are: the mode (*SODE* or *MODE*), the source tasks file, the fitness function to be used (f_1 or f_2), NP, the number of generations, CR, F, the target architecture (set to 'MESH' by default), the number of processor lines, the number or processor columns and the number of grid layers (set to '1' by default), as demonstrated in Figure 5.1.

```
// SO/MO; task file; F1/F2/F3; processors; NP; G; CR; F; MESH; lines; columns; layers;
SO; c:\tasks\cg32x1_v2.cwg; F3; 36; 20; 100; 0.10; 0.10; MESH; 6; 6; 1;
SO; c:\tasks\cg32x1_v2.cwg; F3; 36; 20; 100; 0.20; 0.20; MESH; 6; 6; 1;
SO; c:\tasks\cg32x1_v2.cwg; F3; 36; 20; 100; 0.30; 0.30; MESH; 6; 6; 1;
SO; c:\tasks\cg32x1_v2.cwg; F3; 36; 20; 100; 0.40; 0.40; MESH; 6; 6; 1;
SO; c:\tasks\cg32x1_v2.cwg; F3; 36; 20; 100; 0.50; 0.50; MESH; 6; 6; 1;
SO; c:\tasks\cg32x1_v2.cwg; F3; 36; 20; 100; 0.60; 0.60; MESH; 6; 6; 1;
SO; c:\tasks\cg32x1_v2.cwg; F3; 36; 20; 100; 0.70; 0.70; MESH; 6; 6; 1;
SO; c:\tasks\cg32x1_v2.cwg; F3; 36; 20; 100; 0.80; 0.80; MESH; 6; 6; 1;
```

Figure 5.1: Sample of the input file for the test robot.

5.1 Using the NASA NAS benchmark to evaluate the SODE implementation

The SODE implementation was evaluated by using the NASA NAS Parallel benchmark, as described previously in Section 2.3.1. Test cases were defined by varying the combinations of the execution parameters, as demonstrated in Table 5.1.

Table 5.1: Domain of the Execution parameter values.

Parameters	Range
NP	10 and 20
G	100, 300, 500, 100, 5000 and 10000
CR	0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 06, 0.7, 0.8 and 0.9
F	0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 06, 0.7, 0.8 and 0.9

Since SODE is a stochastic random-based algorithm, a number of independent test cases were created for using different input parameter sets. Each test case was executed at least 30 times in order to reach the required number for consideration as a valid statistical sample. All combinations of input parameters were tried, and special attention was given to the *CR* and *F* pair, making sure they had presented the following ranging behaviours: $CR = \{0.5, 0.1, 0.2, ..., 0.9\}$ while $F = \{0.5, 0.1, 0.2, ..., 0.9\}$; $CR = \{0.5, 0.1, 0.2, ..., 0.9\}$ while $F = \{0.9, 0.8, ..., 0.1, 0.05\}$; and $CR = \{0.9, 0.8, ..., 0.1, 0.05\}$ while $F = \{0.5, 0.1, 0.2, ..., 0.9\}$.

5.1.1 NASA NAS CG Evaluation

The NASA NAS CG simulates a *Conjugate Gradient* calculation application, which is a method used to solve a particular case of linear equations. It is characterized by presenting irregular patterns of memory access and message exchange.

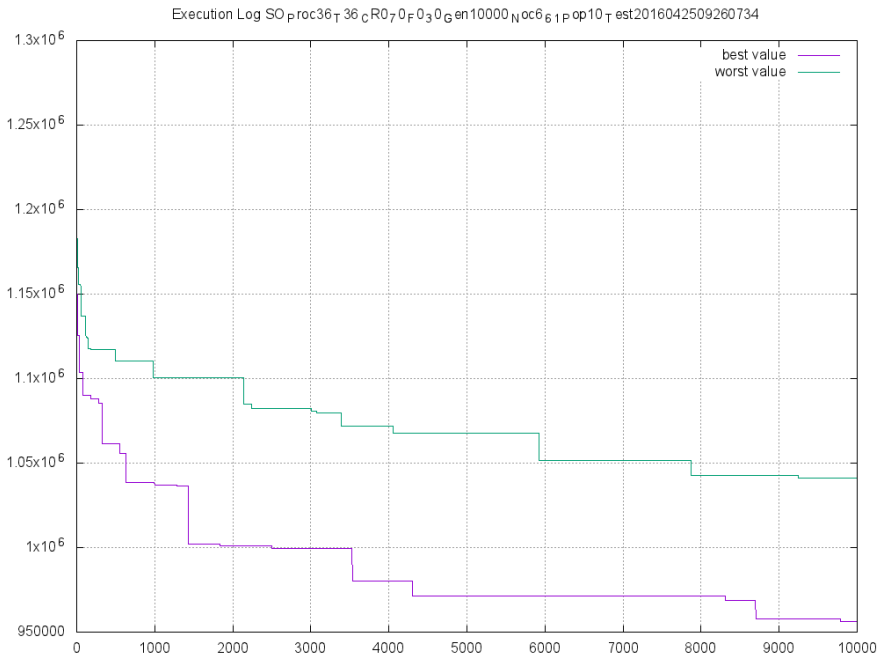


Figure 5.2: Top 5 Best solutions for the NASA NAS CG application: test case 1

Figure 5.2 displays a data plot from the best candidate solution which was generated during the tests using the CG application. This test was executed using the following input parameter set: $NP = 10$, $CR = 0.7$, $F = 0.3$, $G = 1000$. This particular test case took 93009ms to be executed. The line of the worst fitness values (green) suggests the population was subject to a continuing enhancement; where even the line with the best values appeared to be stuck in a particular minimum local (as possible to see from the period that started around generation 4000 to almost generation 8900, when a new minimum was reached). Another important finding was the fact that the algorithm convergence seemed

to be faster on earlier generations; most likely due to the fact that there was more space available to be explored on the solution space when the execution started.

It is possible to find a similar behaviour in the subsequent found best candidate solutions, as demonstrated in the Figures 5.3, 5.4, 5.5 and 5.6.

The plotted test presented in Figure 5.3 used the following input parameter set: NP = 20, CR = 0.7, F = 0.3, G = 1000, and was executed in 242972ms.

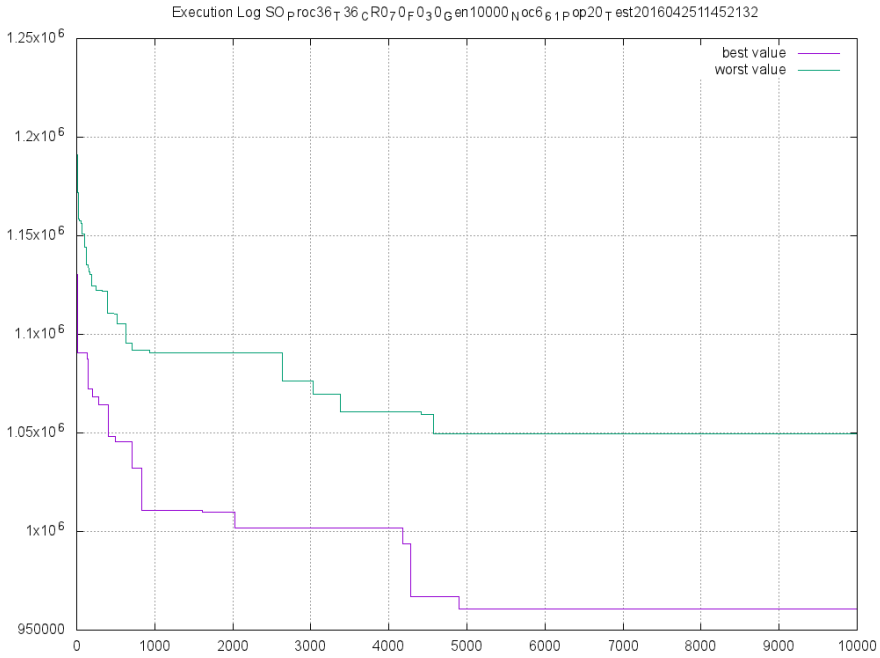


Figure 5.3: Top 5 Best solutions for the NASA NAS CG application: test case 2

The next best candidate solution found had its data plotted in Figure 5.4. This time, the SODE generated a steep decrease in the fitness value immediately in early generations, and kept enhancing this value in subsequent generations; although was not as fast as before. Input parameters were set to NP = 20, CR = 0.8, F = 0.2, G = 10000 and the required execution time was 243395ms.

The following test case is presented in Figure 5.5. At this time, very few enhances were reached throughout the execution. Although, a value few generations obtained a final best value before they reached the stop condition. The following input parameter set was used on this test case: NP = 10, CR = 0.5, F = 0.5, G = 10000 and the execution time was equal to 108016ms.

The final test case amongst the best candidate solutions found, is shown in Figure 5.6. It is possible to see two big enhancements, that were generated throughout the test execution; the first was seen in the beginning generations, and the second appeared around generation number 1700. This test was run in 120726ms and the input parameters were: NP = 20, CR = 0.4, F = 0.4, G = 5000.

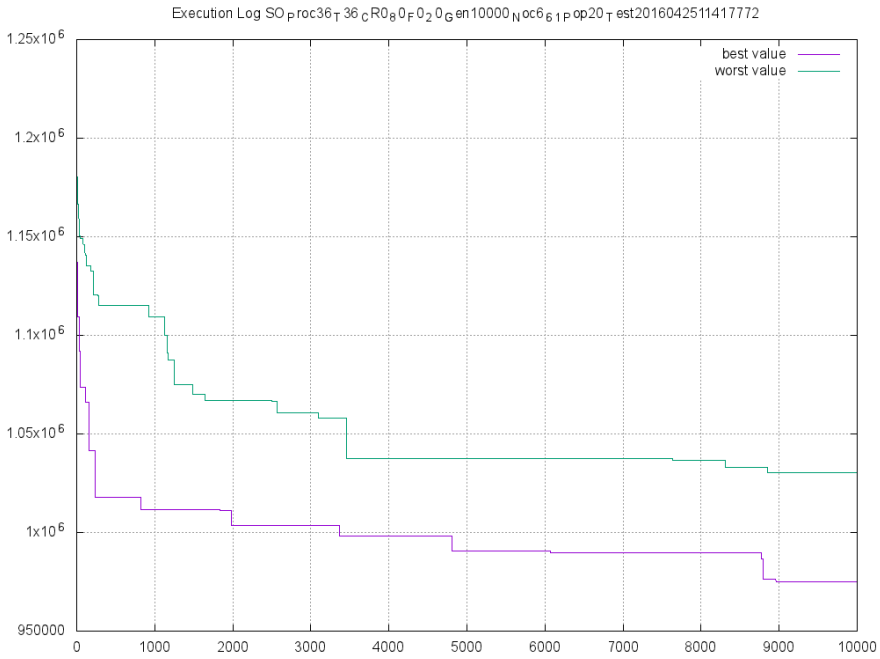


Figure 5.4: Top 5 Best solutions for the NASA NAS CG application: test case 3

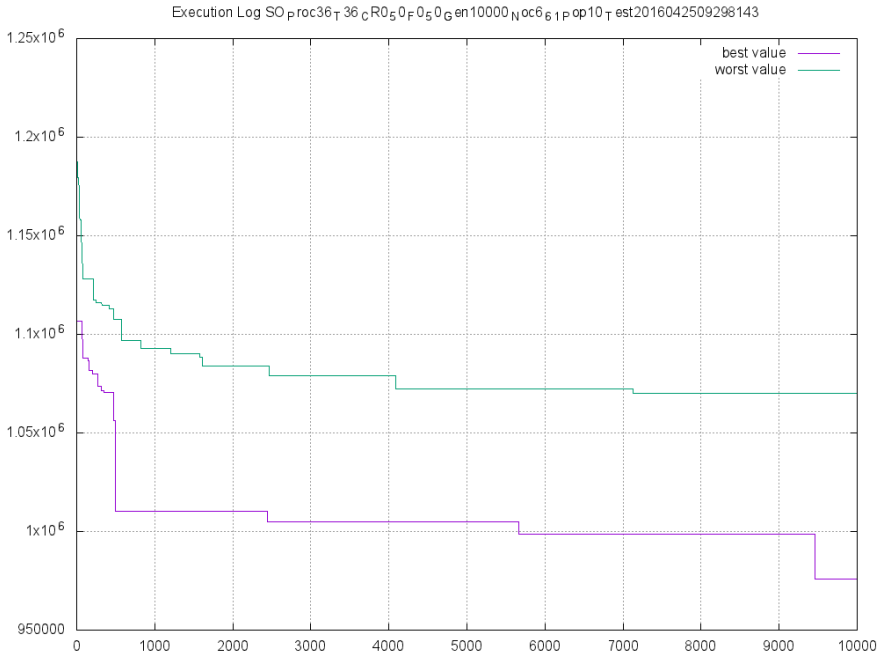


Figure 5.5: Top 5 Best solutions for the NASA NAS CG application: test case 4

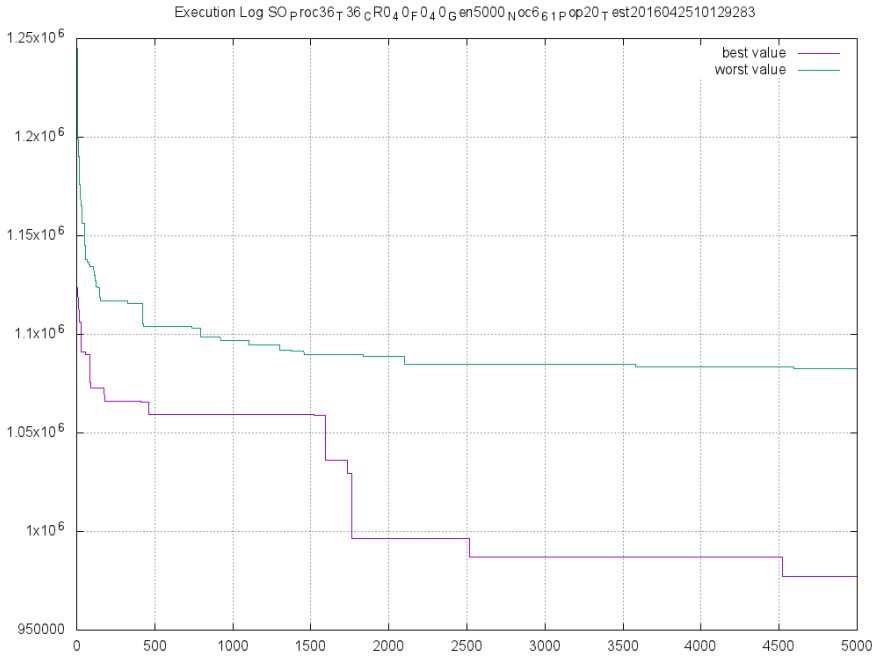


Figure 5.6: Top 5 Best solutions for the NASA NAS CG application: test case 5

5.1.2 Detail Data Results of all tested NASA NAS application benchmarks

All tested applications were evaluated considering two main aspects: the quality of the generated candidate solutions, and the execution time they had consumed to perform.

General Quality of Generated Candidate Solutions

The following tables show the top 5 best candidate solutions found for each NASA NAS benchmark application tested. Data is grouped by the generation size (parameter *G*). The best fitness value found, as well as the fitness mean value and its standard deviation are also shown. This measure is an indication of how the population behaved during the test execution. Smaller values on the standard deviation suggest that a greater number of candidate solutions were able to approach the region where the best solution was located. On the other hand, it was also important to make sure the algorithm was able to explore alternative locations on the solutions space in order to prevent the algorithm to getting stuck into a local minimum area.

Table 5.2 presents the top 5 best values obtained on each group of test case executions, grouping the data by the number of computed generations (G parameter). Table 5.3 does the same for the FT benchmark application. The IS application test results are detailed in Table 5.4; whereas Table 5.5 presents the results of the LU application tests. Finally, Table 5.6 shows the data of the test results obtained after MG application testing.

Table 5.2: Best Solutions using SODE with NASA NAS CG application.

SODE test results running the NASA NAS CG benchmark application											
				Best Fitness Generation				Best Fitness Value			
G	NP	CR	F	Mean	Best	Worst	Stdev	Mean	Best	Worst	Stdev
100	20	0.2	0.2	63	19	97	29	1789528	1037680	2001172	194267
100	10	0.4	0.4	72	15	97	29	1898272	1040564	2057930	224526
100	20	0.6	0.4	63	11	98	33	1849517	1047111	2044072	227339
100	20	0.4	0.4	46	10	90	26	1870105	1062317	2004180	212459
100	20	0.4	0.6	63	12	98	30	1848679	1078681	1983915	225329
300	10	0.4	0.4	222	109	296	63	1791844	1046383	1928953	216210
300	10	0.4	0.6	251	151	294	32	1829180	1050756	1956887	212704
300	20	0.8	0.2	154	10	277	105	1720465	1051436	1883414	179233
300	10	0.7	0.3	209	54	298	80	1800307	1052310	1934368	204320
300	20	0.4	0.6	140	11	285	94	1784007	1060554	1918935	194512
500	20	0.6	0.4	248	22	459	155	1756861	1026169	1924702	217243
500	10	0.2	0.2	293	10	499	191	1695562	1038611	1934008	172524
500	20	0.5	0.5	332	21	499	134	1793411	1039000	1927421	162356
500	20	0.9	0.1	274	26	497	183	1442714	1040603	1710185	203817
500	10	0.7	0.3	248	10	479	154	1775265	1041377	1876809	181022
1000	10	0.6	0.6	634	220	981	256	1764817	996990	1908720	212729
1000	20	0.7	0.3	671	38	956	281	1686346	1002204	1853708	211006
1000	10	0.8	0.2	640	13	983	369	1619631	1014140	1992747	207921
1000	20	0.8	0.2	533	11	990	400	1662831	1027161	1860116	178468
1000	20	0.5	0.5	601	10	999	339	1756503	1027614	1906023	148314
5000	20	0.4	0.4	2626	13	4964	1737	1555924	977349	1858030	191064
5000	20	0.5	0.5	3306	15	4984	1482	1632847	992284	1958477	145583
5000	20	0.8	0.2	2800	10	4989	2168	1498114	992845	1862494	269410
5000	10	0.6	0.6	3221	213	4988	1577	1651905	993967	1783609	185865
5000	20	0.7	0.3	2656	10	4959	1833	1615491	994603	1956217	211961
10000	10	0.7	0.3	6555	11	9864	5667	1434127	956425	1937284	490925
10000	20	0.7	0.3	2011	18	5023	2692	1579205	960702	1811529	352678
10000	20	0.8	0.2	1215	12	8959	2964	1635647	975035	1903899	350996
10000	10	0.5	0.5	7091	5266	9460	1934	1296877	975672	1624648	348499
10000	20	0.5	0.5	3194	11	9580	3713	1540025	977738	1884257	369674

Table 5.3: Best Solutions using SODE with NASA NAS FT application.

SODE test results running the NASA NAS FT benchmark application											
				Best Fitness Generation				Best Fitness Value			
G	NP	CR	F	Mean	Best	Worst	Stdev	Mean	Best	Worst	Stdev
100	20	0.9	0.1	45	12	97	30	3061574	2902705	3104730	64920
100	20	0.7	0.3	46	12	81	49	2994473	2902891	3086055	129517
100	10	0.4	0.6	45	10	81	50	3017030	2919890	3114170	137377
100	10	0.7	0.7	34	13	56	30	3015335	2923582	3107088	129758
100	10	0.1	0.1	48	12	99	31	3077003	2928283	3123282	54323
300	20	0.7	0.3	63	13	204	94	3016365	2814974	3106117	135361
300	10	0.8	0.2	138	16	261	173	2962427	2830491	3094363	186586
300	20	0.9	0.1	142	30	276	83	3061405	2851269	3097316	70065
300	10	0.3	0.3	53	13	93	57	2989329	2854231	3124427	191057
300	20	0.5	0.5	165	80	250	120	2898833	2873235	2924430	36200
500	20	0.5	0.5	127	11	307	141	2942947	2719051	3100288	186746
500	20	0.8	0.2	47	10	146	66	3025981	2793280	3124047	155889
500	20	0.9	0.1	121	12	288	94	3051766	2802796	3098035	93989
500	10	0.7	0.3	128	12	244	164	2973508	2823874	3123142	211614
500	20	0.7	0.3	164	11	467	262	3025147	2826896	3124657	171691
1000	20	0.6	0.4	109	14	392	188	2994175	2722507	3092367	181196
1000	20	0.7	0.3	331	10	955	540	2990424	2769662	3123203	192493
1000	20	0.5	0.5	348	10	921	460	2994846	2788301	3107088	152165
1000	20	0.6	0.6	481	28	934	641	2937987	2788831	3087143	210938
1000	20	0.4	0.4	399	15	784	544	2957224	2790626	3123821	235604
5000	10	0.7	0.3	656	12	2584	1285	2981251	2598294	3124865	256262
5000	20	0.7	0.3	1197	26	4684	2325	2971038	2616395	3106865	236756
5000	10	0.5	0.5	2449	23	4165	2161	2805819	2632472	3095754	252690
5000	20	0.8	0.2	673	10	3954	1607	3022432	2658857	3124853	178878
5000	20	0.4	0.6	204	10	398	274	2886045	2695581	3076508	269356
10000	20	0.5	0.5	3495	13	7509	4045	2870212	2618763	3116888	265030
10000	20	0.4	0.4	2681	16	8008	4613	2926434	2620444	3085381	265062
10000	10	0.5	0.5	8195	6960	9431	1747	2678982	2628470	2729493	71434
10000	20	0.7	0.3	2922	10	8747	5044	2943917	2640106	3107088	263349
10000	20	0.8	0.2	2213	10	8822	4406	2990663	2644067	3116638	231194

Table 5.4: Best Solutions using SODE with NASA NAS IS application

SODE test results running the NASA NAS IS benchmark application											
				Best Fitness Generation				Best Fitness Value			
G	NP	CR	F	Mean	Best	Worst	Stdev	Mean	Best	Worst	Stdev
100	20	0,9	0,1	50	25	99	28	1131270	1125246	1138435	5022
100	10	0,3	0,3	12	10	16	3	1139102	1128207	1146510	9637
100	20	0,1	0,1	44	12	81	30	1132704	1130540	1134266	1327
100	10	0,05	0,05	45	11	89	28	1137859	1130619	1148180	5301
100	10	0,9	0,1	37	16	89	25	1138914	1130696	1144822	5164
300	10	0,1	0,1	22	11	59	18	1131790	1122616	1139414	5724
300	20	0,3	0,3	13	11	16	3	1134579	1124307	1141733	9122
300	20	0,9	0,1	72	19	147	49	1131712	1125493	1138520	4345
300	20	0,05	0,05	86	21	212	60	1133797	1128011	1137946	2749
300	10	0,8	0,2	15	13	17	3	1131721	1130341	1133100	1951
500	20	0,9	0,1	46	14	87	31	1133790	1124764	1139017	5192
500	20	0,1	0,1	90	42	251	80	1132870	1126486	1137554	4142
500	10	0,05	0,05	82	12	250	69	1136304	1126700	1149241	5077
500	20	0,7	0,3	27	12	44	16	1132888	1127176	1139456	6185
500	10	0,1	0,1	68	26	127	40	1134628	1128001	1139478	3846
1000	10	0,9	0,1	35	15	59	14	1135542	1122616	1145291	7889
1000	20	0,9	0,1	107	35	230	62	1129923	1125766	1133717	3301
1000	20	0,1	0,1	37	10	86	29	1130375	1127043	1133521	2652
1000	20	0,8	0,2	28	11	41	16	1132463	1127105	1137891	5393
1000	20	0,05	0,05	162	11	544	150	1134438	1130062	1139018	2572
5000	20	0,8	0,2	36	19	67	27	1131388	1123671	1137843	7170
5000	20	0,9	0,1	100	14	262	99	1131491	1125613	1135899	4110
5000	20	0,1	0,1	64	10	145	50	1131274	1126699	1135616	2614
5000	20	0,05	0,05	150	22	446	141	1134272	1128046	1143326	3392
5000	20	0,3	0,3	22	15	33	10	1133918	1131945	1136983	2690
10000	20	0,3	0,3	31	14	48	24	1133843	1126174	1141511	10845
10000	20	0,05	0,05	120	17	449	107	1133930	1126892	1138887	2895
10000	20	0,1	0,1	69	32	122	38	1130532	1126895	1134240	2865
10000	20	0,7	0,3	25	24	27	2	1134519	1127176	1139261	6449
10000	10	0,9	0,1	31	19	52	11	1135792	1131023	1138362	2882

Table 5.5: Best Solutions using SODE with NASA NAS LU application.

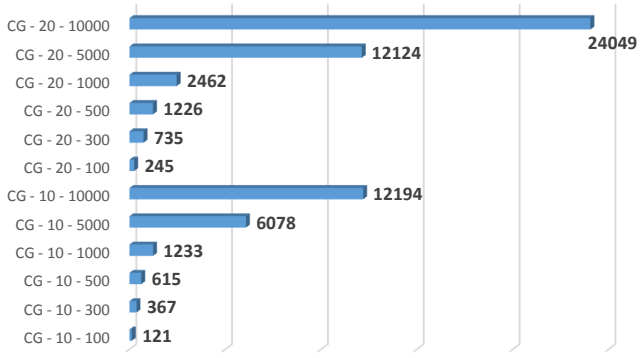
SODE test results running the NASA NAS LU benchmark application											
				Best Fitness Generation				Best Fitness Value			
G	NP	CR	F	Mean	Best	Worst	Stdev	Mean	Best	Worst	Stdev
100	20	0.9	0.1	83	69	96	11	4114793	3947629	4306354	147461
100	10	0.1	0.1	83	70	95	10	4284131	4079869	4433972	149596
100	10	0.05	0.05	94	89	99	4	4512482	4320080	4680297	148082
100	20	0.8	0.2	82	68	96	11	4956674	4720959	5089371	167115
100	10	0.9	0.1	54	43	65	9	4934605	4734089	5083388	147218
300	20	0.9	0.1	176	158	193	14	3896750	3721676	4107653	159598
300	20	0.1	0.1	159	143	175	13	4347353	4130250	4514092	160699
300	10	0.05	0.05	194	173	214	17	4342720	4188618	4546978	150537
300	10	0.1	0.1	69	46	92	19	4587617	4421274	4792477	153971
300	20	0.4	0.4	47	27	66	16	4753508	4522150	4881232	163890
500	20	0.9	0.1	178	94	261	68	4015476	3828909	4206471	154171
500	20	0.1	0.1	225	136	178	73	4198977	4020634	4399109	155278
500	10	0.05	0.05	130	21	240	155	4622469	4257177	4987761	516601
500	10	0.9	0.1	130	29	230	82	4634355	4425601	4777657	150994
500	10	0.1	0.1	214	104	324	90	4626515	4467360	4835815	154544
1000	20	0.05	0.05	382	315	449	95	4242671	4057703	4427639	261584
1000	20	0.1	0.1	160	116	204	36	4269924	4104967	4493150	163744
1000	20	0.9	0.1	98	50	146	39	4592006	4429736	4797826	153395
1000	20	0.2	0.2	80	38	122	34	5004842	4798761	5156338	150997
1000	10	0.7	0.3	101	57	145	36	4988517	4826846	5174964	143194
5000	20	0.9	0.1	146	89	202	46	4142872	3974924	4322387	142087
5000	20	0.1	0.1	83	22	144	50	4209548	4007302	4377772	153145
5000	20	0.05	0.05	124	79	170	64	4456883	4387956	4525810	97477
5000	10	0.9	0.1	146	87	204	48	4706665	4517852	4898434	155386
5000	20	0.4	0.4	79	18	140	50	4797465	4594562	4965337	153382
10000	20	0.05	0.05	198	45	352	217	4302870	3818578	4787161	684892
10000	10	0.05	0.05	149	66	232	117	4219317	4055089	4383545	232253
10000	10	0.1	0.1	271	70	471	164	4317691	4082057	4441460	166689
10000	20	0.9	0.1	292	128	456	134	4410640	4218857	4588729	151310
10000	20	0.1	0.1	214	41	386	141	4633070	4466170	4845085	157946

Table 5.6: Best Solutions using SODE with NASA NAS MG application.

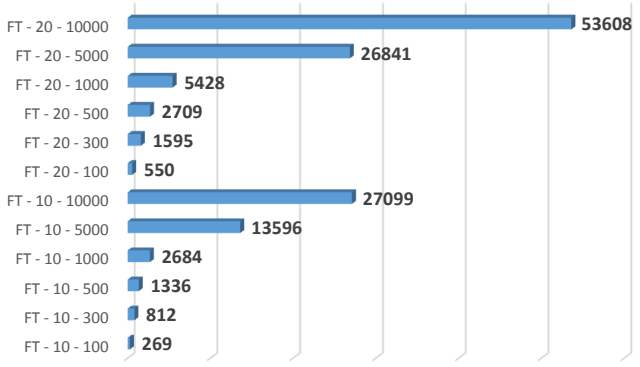
SODE test results running the NASA NAS MG benchmark application											
				Best Fitness Generation				Best Fitness Value			
G	NP	CR	F	Mean	Best	Worst	Stdev	Mean	Best	Worst	Stdev
100	20	0.1	0.1	54	41	75	57	681934	681792	682075	116
100	10	0.1	0.1	24	10	29	21	694506	694272	694739	191
100	20	0.05	0.05	77	62	92	21	709529	709259	709799	382
100	10	0.9	0.1	22	10	27	20	719083	718884	719282	162
100	20	0.9	0.1	69	54	96	73	721410	721249	721571	131
300	20	0.05	0.05	259	249	270	15	645986	636021	655951	14093
300	20	0.7	0.3	59	48	83	63	680206	674706	685706	4491
300	10	0.9	0.1	43	21	54	39	683718	677545	689891	5040
300	20	0.1	0.1	168	119	228	172	691035	685307	696763	4677
300	10	0.05	0.05	219	141	297	110	709846	693146	726547	23618
500	20	0.05	0.05	181	71	292	156	670529	645793	695265	34982
500	20	0.1	0.1	179	155	257	197	665993	651611	680375	11743
500	20	0.8	0.2	41	30	56	42	695418	681496	709339	11367
500	10	0.1	0.1	145	131	211	162	700733	685796	715670	12196
500	20	0.9	0.1	111	96	159	122	712433	696939	727926	12650
1000	10	0.9	0.1	91	72	127	97	678438	674490	682385	3223
1000	10	0.1	0.1	78	55	106	80	689363	684896	693830	3647
1000	20	0.1	0.1	95	83	137	105	696248	691316	701179	4027
1000	20	0.05	0.05	183	171	195	17	697418	692958	701878	6307
1000	20	0.9	0.1	70	55	98	74	700684	695702	705666	4068
5000	20	0.05	0.05	186	89	284	138	681737	668963	694512	18066
5000	10	0.05	0.05	43	32	55	16	700506	690789	710224	13743
5000	20	0.9	0.1	115	100	165	127	705997	693560	718434	10155
5000	10	0.9	0.1	176	160	256	197	709242	696766	721718	10187
5000	20	0.1	0.1	36	14	43	31	711610	697678	725541	11375
10000	10	0.05	0.05	89	73	106	23	694470	674718	714223	27934
10000	20	0.9	0.1	118	95	166	126	699399	680927	717870	15082
10000	20	0.05	0.05	111	54	169	81	712112	701177	723048	15465
10000	20	0.1	0.1	122	73	159	118	719215	701289	737141	14637
10000	10	0.9	0.1	146	109	201	152	729683	710301	749065	15825

Execution Time

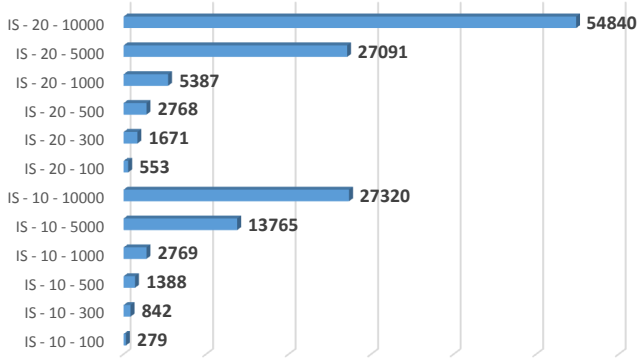
This section presents the results obtained from recording the execution times on all performed tests.



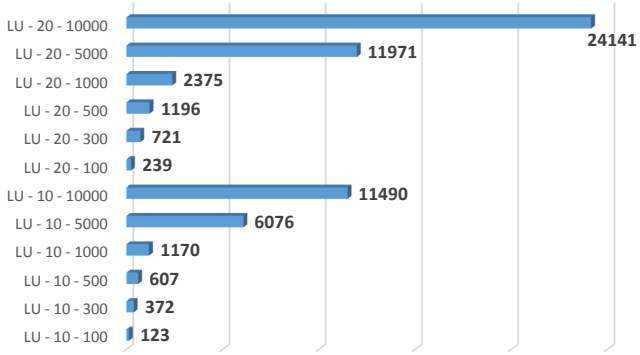
(a) NASA NAS CG application.



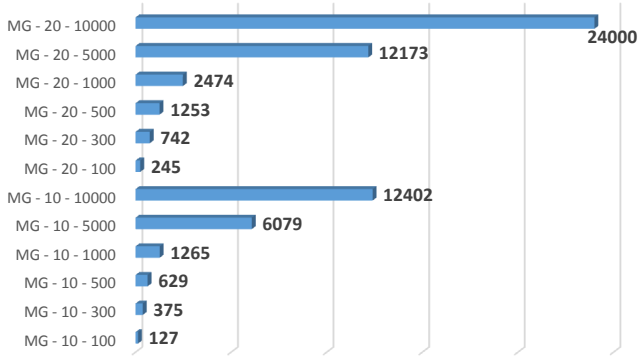
(b) NASA NAS FT application.



(c) NASA NAS IS application.



(d) NASA NAS LU application.



(e) NASA NAS MG application.

Figure 5.7: Execution time of NASA NAS application benchmarks running under the SODE implementation.

Figure 5.7 presents the data plots from execution times taken during the test executions. It is possible to see the execution time grow at almost the same rate as the Population, times the number of generations ($G * NP$). Although all tested applications seemed to have behaved the same way in regards to the growth of execution time measured, each application presented particular values for its execution time, as demonstrated in Figure 5.8.

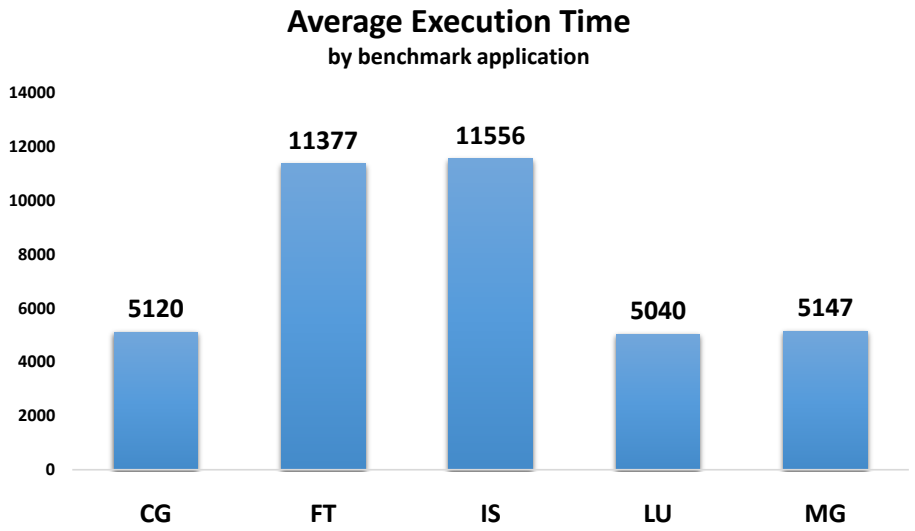


Figure 5.8: Global comparison of execution time.

Despite all tested applications having the exactly same number of tasks to compute, the number of send and receive messages amongst them were particular to each application. For example, the FT application, which implements an 3D fast Fourier Transform, performs all-to-all communication; a fact that was reflected in the extra time required to compute its benchmark.

5.1.3 SODE versus CAFES Comparison

This section shows the results of a comparison executed between the CAFES framework and our proposed implementation. The comparison was based on the quality of the candidate solutions generated by both approaches.

Table 5.7 shows a frame comparing the top 5 best candidate solutions generated by each implementation, SODE and CAFES.

The SODE implementation was superior to CAFES in generating better candidate solutions for the CG and FT applications; however, for the IS application, solutions generated for the CAFES framework were slightly more efficient (around 1% better) than those generated by the SODE. Finally, CAFES was superior in generating candidate solutions for the LU and MG applications, as shown in Figure 5.9.

Another important indication of a good convergence is the standard deviation metric. It measures the proximity of the best candidate solutions to each other in the solution space, and may suggest how far they are from an optimal solution. Although, it is not able to predict whether or not it is a global or local optimum region. Figure 5.10 shows a comparative between the two tested approaches.

Table 5.7: Comparison between SODE and CAFES generated solutions.

		Best Fitness Value		
Bench	**App**	**Mean**	**Top 5**	**Stdev**
CG	**SODE**	969114	956425 960702 975035 975672 977738	8766
	CAFES	989330	975064 977700 992170 995720 1005998	11537
FT	**SODE**	2616473	2598294 2616395 2618763 2620444 2628470	9954
	CAFES	3020149	3020280 3020140 3019975 3020201 3020147	100
IS	**SODE**	1124121	1122616 1123671 1124307 1124764 1125246	914
	CAFES	1109178	1108580 1108897 1108869 1110224 1109320	574

		Best Fitness Value		
Bench	**App**	**Mean**	**Top 5**	**Stdev**
LU	**SODE**	3858343	3721676 3818578 3828909 3947629 3974924	92400
	CAFES	2503478	2546819 2401693 2523786 2522545 2522545	51726
MG	**SODE**	655376	636021 645793 651611 668963 674490	14357
	CAFES	485965	490259 483629 485791 492037 478110	4950

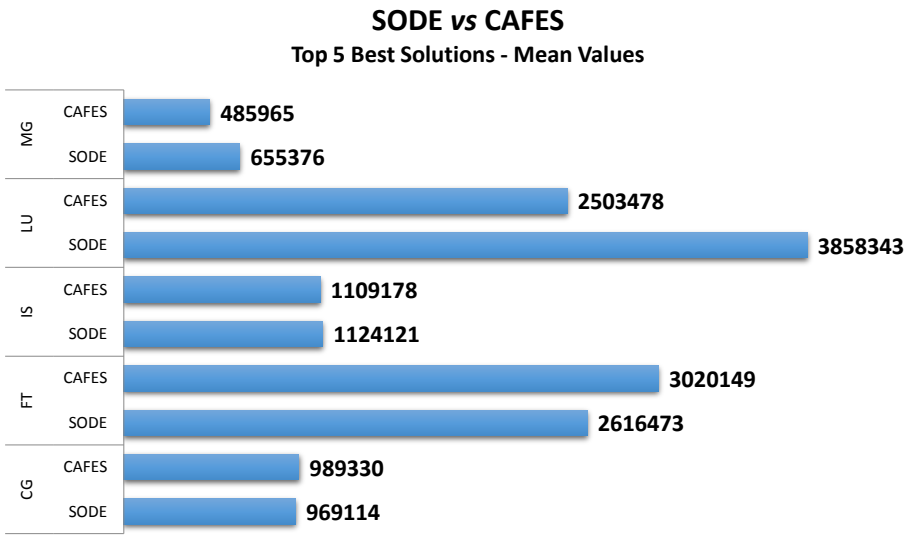


Figure 5.9: SODE vs CAFES: quality of generated candidate solutions.

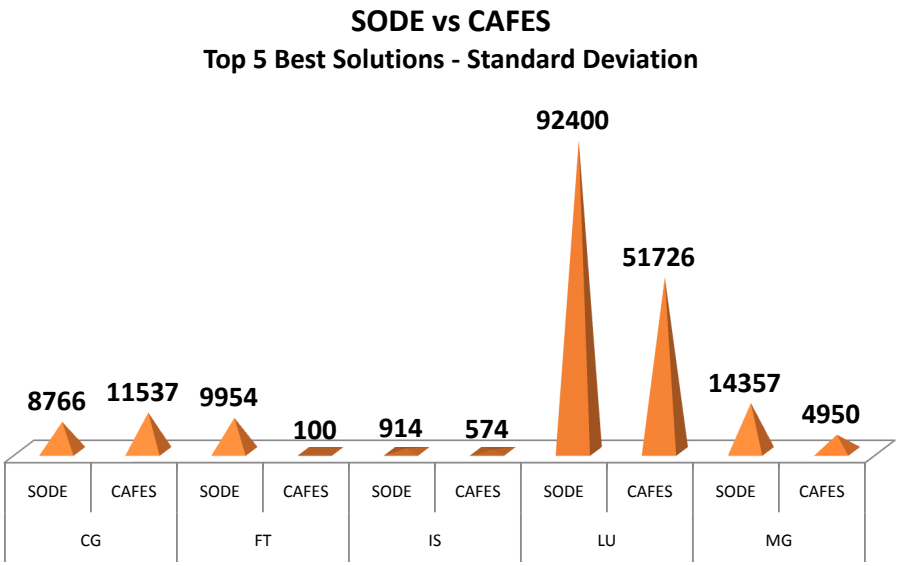


Figure 5.10: SODE vs CAFES: standard deviation comparison.

The SODE implementation was more efficient on keeping a small standard deviation when computing the CG application benchmark. However, for all other tested applications, CAFES was superior on reaching smaller values for this metric.

5.2 Conclusions

Effectiveness of the proposed SODE implementation was demonstrated as a result of the experiments based on the NASA NAS Parallel and its select benchmark applications. Our implementation proved to keep the good qualities of convergence of the classical DE implementations. The importance of choosing the right set of input parameter for each applications was also demonstrated. Finally, our SODE implementation has shown it can be considered as a viable alternative to the CAFES framework for some applications.

6. CONCLUSIONS

The Task Mapping onto NoC is a NP-Hard class problem, which means that brute force approaches are not viable to solve this class of problems. For this reason, heuristic methods are frequently used to help solve this challenge. Evolutionary Algorithms represent one important branch within heuristic search techniques; and amongst these branches rests the Differential Evolution algorithm.

In this work we have proposed an innovative approach for solving the Task Mapping onto Noc Problem by using a SODE implementation. Our algorithm extended the classical SODE by adding a new procedure inside the DE's Mutation genetic operator, which started to reward candidate solutions that contained a seed, indicative of a previously identified relationship amongst communicating tasks. Since the seed identification was executed only once, and before it started computing the DE generations, its impact on the asymptotic complexity of the main computing algorithm was negligible.

Our implementation was evaluated by running the classical NASA NAS Parallel benchmark, and by selecting the applications within this package that relied on task communication. The chosen applications were CG, FT, IS, LU and MG. It was demonstrated that our SODE implementation was able to generate feasible candidate solutions for these benchmark applications, provided that the right combination of input parameters was made. For this reason, tests were run using different combinations of input parameters, trying to find the best combination for them in each tested application.

Tests have shown our implementation was superior than the CAFES framework for computing CG and FT application benchmarks, similar (around 1% less efficient) for the IS application and less efficient for the LU and MG applications.

These results have shown the importance of the proposed algorithm for solving the Task Mapping problem, especially when such applications have a similar profile to those where our implementation was superior.

This work can potentially contribute to future implementations based on the parallel DE; possibly seeking to explore more fronts of the space solutions by using concurrent agents, but exploring the same united Population set.

Finally, this work has made a contribution to improving the CAFES framework, by exploring its code and mapping opportunities for future enhancements to be made on that tool set.

References

[1] Al-Wattar, A.; Areibi, S.; Grewal, G. “Efficient mapping and allocation of execution units to task graphs using an evolutionary framework”, *ACM SIGARCH Computer Architecture News*, vol. 43–4, Sep 2016, pp. 46–51.

[2] Almojel, A. “Characterization of ilp distribution for nasa nas parallel benchmarks”, *Journal of King Saud University - Computer and Information Sciences*, vol. 16, Jan 2004, pp. 45–65.

[3] Antunes, E.; Aguiar, A.; et al.. “Partitioning and mapping on NoC-based MPSoC: an energy consumption saving approach”. In: International Workshop on Network on Chip Architectures (NoCArc), 2011, pp. 51–56.

[4] Bao, Y.; Bienia, C.; Li, K. “The PARSEC benchmark suite tutorial”. Source: http://parsec.cs.princeton.edu/download/tutorial/3.0/parsec-tutorial.pdf, July 2016.

[5] Bienia, C.; Kumar, S.; et al.. “The PARSEC benchmark suite: Characterization and architectural implications”. In: International Conference on Parallel Architectures and Compilation Techniques (PACT), 2008, pp. 72–81.

[6] Bokhari, S. “A shortest tree algorithm for optimal assignments across space and time in a distributed processor system”, *Transactions on Software Engineering*, vol. 7–6, Nov 1981, pp. 583–589.

[7] Braun, T.; Siegel, H., N.; et al.. “A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems”, *Journal of Parallel and Distributed computing*, vol. 61–6, Jun 2001, pp. 810–837.

[8] Chen, W. “Task partitioning and mapping algorithms for multi-core packet processing systems”, Ph.D. Thesis, University of Massachusetts Amherst, 2009, 72p.

[9] Chu, W. “Optimal file allocation in a multiple computer system”, *Transactions on Computers*, vol. C-18–10, Oct 1969, pp. 885–889.

[10] Cortes, O.; Pais, M.; et al.. “Differential evolution on a gpgpu: The influence of parameters on speedup and the quality of solutions”. In: International Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015, pp. 299–306.

[11] Dally, W. J.; Towles, B. “Route packets, not wires: on-chip interconnection networks”. In: Design Automation Conference (DAC), 2001, pp. 684–689.

[12] Das, D.; Verma, L.; Das, A. “A differential evolutionary approach to solve the hardware software partitioning problem”, *International Journal of Engineering Research and Technology*, vol. 3–7, Jul 2016, pp. 5.

[13] Das, S.; Suganthan, P. N. “Differential evolution: A survey of the state-of-the-art”, *Transactions on Evolutionary Computation*, vol. 15–1, Feb 2011, pp. 4–31.

[14] Deng, C.; Zhao, B.; et al.. “Modified differential evolution for task assignment problem”. In: International Workshop on Intelligent Systems and Applications (ISA), 2010, pp. 1–4.

[15] Dick, R.; Rhodes, D.; Wolf, W. “Tgff: task graphs for free”. In: International Workshop on Hardware/Software Codesign (CODES/CASHE), 1998, pp. 97–101.

[16] Division, N. A. S. “NAS parallel benchmarks”. Source: http://www.nas.nasa.gov/publications/npb.html, Jul 2016.

[17] Fisher, N.; Anderson, J.; Baruah, S. “Task partitioning upon memory-constrained multiprocessors”. In: Embedded and Real-Time Computing Systems and Applications (RTCSA), 2005, pp. 416–421.

[18] Fister, I.; Fister Jr, I.; et al.. “A comprehensive review of firefly algorithms”, *Swarm and Evolutionary Computation*, vol. 13, Dec 2013, pp. 34–46.

[19] for Machine Learning, C.; Systems, I. “UCI machine learning repository”. Source: http://archive.ics.uci.edu/ml/, Jul 2007.

[20] Göhringer, D.; Hübner, M.; et al.. “A design methodology for application partitioning and architecture development of reconfigurable multiprocessor systems-on-chip”. In: International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2010, pp. 259–262.

[21] Goldberg, D. “Genetic Algorithms in Search, Optimization and Machine Learning”. Addison-Wesley Longman Publishing Co., Inc., 1989, 1 ed., 432p.

[22] Haeser, G.; Ruggiero, M. “Aspectos teóricos de simulated annealing e um algoritmo duas fases em otimização global”, *Trends in Applied and Computational Mathematics*, vol. 9–3, Sep 2008, pp. 395–404.

[23] Hao, K.; Wang, B.; Luo, Y. “Multi-objective network coding optimization based on NSGA-ii algorithm”. In: International Conference on Control Engineering and Communication Technology (ICCECT), 2012, pp. 843–846.

[24] Heineman, G.; Pollice, G.; Selkow, S. “Algorithms in a nutshell: a practical guide”. O’Reilly Media, Inc., 2016, 2 ed., 375p.

[25] Hu, J.; Marculescu, R. “Energy-aware mapping for tile-based NoC architectures under performance constraints”. In: Asia and South Pacific Design Automation Conference (ASP-DAC), 2003, pp. 233–239.

[26] Ingber, L. “Simulated annealing: Practice versus theory”, *Mathematical and computer modelling*, vol. 18–11, Dec 1993, pp. 29–57.

[27] Jena, R.; Sharma, G. “A multiobjective evolutionary algorithm-based optimisation model for network on chip synthesis”, *International Journal of Innovative Computing and Applications*, vol. 1–2, Jan 2007, pp. 121–127.

[28] Kirkpatrick, S.; Gelatt, C.; Vecchi, M. “Optimization by simulated annealing”, *Science*, vol. 220–4598, May 1983, pp. 671–680.

[29] Krömer, P.; Platoš, J.; et al.. “An implementation of differential evolution for independent tasks scheduling on GPU”. In: International Conference on Hybrid Artificial Intelligence Systems (HAIS), 2011, pp. 372–379.

[30] Le Beux, S.; Bois, G.; et al.. “Combining mapping and partitioning exploration for noc-based embedded systems”, *Journal of Systems Architecture*, vol. 56–7, Jul 2010, pp. 223–232.

[31] Lim, K.; Ibrahim, Z.; et al.. “Improving vector evaluated particle swarm optimisation by incorporating nondominated solutions”, *The Scientific World Journal*, vol. 2013–1, Mar 2013, pp. 19.

[32] Linden, R. “Algoritmos Genéticos”. BRASPORT, 2008, 2 ed., 400p.

[33] Mandelli, M. “Exploration of runtime distributed mapping techniques for emerging large scale MPSoCS”, Ph.D. Thesis, Pontifícia Universidade Católica do Rio Grande do Sul, 2015, 134p.

[34] Maravilha, A.; Ramírez, J.; Campelo, F. “Combinatorial optimization with differential evolution: a set-based approach”. In: Conference on Genetic and Evolutionary Computation (GECCO), 2014, pp. 69–70.

[35] Marcon, C. “Modelos para o mapeamento de aplicações em infra-estruturas de comunicação intrachip”, Ph.D. Thesis, Universidade Federal do Rio Grande do Sul, 2005, 192p.

[36] Marcon, C.; Calazans, N.; et al.. “Exploring NoC mapping strategies: an energy and timing aware technique”. In: Conference on Design, Automation and Test in Europe (DATE), 2005, pp. 502–507.

[37] Marcon, C.; Calazans, N.; et al.. “CAFES: A framework for intrachip application modeling and communication architecture design”, *Journal of Parallel and Distributed Computing*, vol. 71–5, May 2011, pp. 714–728.

[38] Mishra, K.; Harit, S. “A fast algorithm for finding the non dominated set in multi objective optimization”, *International Journal of Computer Applications*, vol. 1–25, Feb 2010, pp. 35–39.

[39] Nedjah, N.; Da Silva, M.; Mourelle, L. “Customized computer-aided application mapping on NoC infrastructure using multi-objective optimization”, *Journal of Systems Architecture*, vol. 57–1, Jan 2011, pp. 79–94.

[40] of Industrial Engineering, F.; Management. “The cross-entropy method”. Source: http://iew3.technion.ac.il/CE/, Jun 2016.

[41] Ogras, U.; Hu, J.; Marculescu, R. “Key research problems in noc design: a holistic perspective”. In: International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2005, pp. 69–74.

[42] Price, K.; Storn, R.; Lampinen, J. “Differential evolution: a practical approach to global optimization”. Springer Science & Business Media, 2006, 1 ed., 540p.

[43] Qingqi, Z.; Yanling, Q.; et al.. “Multi-objective mapping for network-on-chip based on bio-inspired optimization algorithms”. In: Prognostics and System Health Management Conference (PHM), 2014, pp. 387–390.

[44] Rechenberg, I. “Evolutionsstrategie : Optimierung technischer Systeme nach Prinzipien der biologischen Evolution”. Frommann-Holzboog, 1973, 1 ed., 161p.

[45] Roy, A.; Manna, K.; Chattapadhay, S. “Effect of core ordering on application mapping onto mesh based network-on-chip design”. In: International Conference on Computing for Sustainable Global Development (INDIACom), 2015, pp. 363–369.

[46] Sahu, P.; Manna, K.; et al.. “Extending kernighan–lin partitioning heuristic for application mapping onto network-on-chip”, *Journal of Systems Architecture*, vol. 60–7, Aug 2014, pp. 562–578.

[47] Saini, S.; Bailey, D. “NAS parallel benchmark (version 1.0) results 11-96. performance comparison of HPF and MPI based NAS parallel benchmarks”. In: NASA Ames Research Center, 1997, pp. 53.

[48] Shen, C.; Tsai, W. “A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion”, *Transactions on Computers*, vol. C-34–3, Mar 1985, pp. 197–203.

[49] Singh, A. K.; Srikanthan, T.; et al.. “Communication-aware heuristics for run-time task mapping on NoC-based MPSoC platforms”, *Journal of Systems Architecture*, vol. 56–7, Jul 2010, pp. 242–255.

[50] Southern, G.; Renau, J. “Deconstructing PARSEC scalability”. In: Workshop on Duplicating, Deconstructing, and Debunking (WDDD), 2015, pp. 10.

[51] Srinivasan, M.; De Micheli, G. “Bandwidth-constrained mapping of cores onto NoC architectures”. In: Conference on Design, Automation and Test in Europe (DATE), 2004, pp. 896–901.

[52] Storn, R.; Price, K. “Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces”, *Journal of global optimization*, vol. 11–4, Dec 1997, pp. 341–359.

[53] Storn, R.; Price, K. “Differential evolution (de) for continuous function optimization (an algorithm by kenneth price and rainer storn)”. Source: http://www1.icsi.berkeley.edu/~storn/code.html, Jul 2016.

[54] Umamaheswari, S.; Kirthiga, K.; et al.. “Cost aware task scheduling and core mapping on network-on-chip topology using firefly algorithm”. In: International Conference on Recent Trends in Information Technology (ICRTIT), 2013, pp. 657–662.

[55] Wah, B.; Lien, Y. “Design of distributed databases on local computer systems with a multiaccess network”, *Transactions on Software Engineering*, vol. SE-11–7, Jul 1985, pp. 606–619.

[56] Waheed, A.; Yan, J. “Workload characterization of cfd applications using partial differential equation solvers”. In: Workshop on Workload Characterization in High-Performance Computing Environments, 1998, pp. 35.

[57] Walter, I.; Cidon, I.; et al.. “The era of many-modules soc: revisiting the noc mapping problem”. In: International Workshop on Network on Chip Architectures (NoCArc), 2009, pp. 43–48.

[58] Wu, M.; Karkar, A.; et al.. “Network on chip optimization based on surrogate model assisted evolutionary algorithms”. In: Congress on Evolutionary Computation (CEC), 2014, pp. 3266–3271.

[59] Xue, B.; Fu, W.; Zhang, M. “Differential evolution (de) for multi-objective feature selection in classification”. In: Conference on Genetic and Evolutionary Computation (GECCO), 2014, pp. 83–84.

[60] Zhao, S.; Hao, Z.; et al.. “Multi-objective differential evolution algorithm based on adaptive mutation and partition selection.”, *Journal of Computers*, vol. 8–10, Oct 2013, pp. 2695–2700.

[61] Zielinski, K.; Laur, R. “Constrained single-objective optimization using differential evolution.” In: Congress on Evolutionary Computation (CEC), 2006, pp. 223–230.

[62] Zitzler, E. “Density and approximations of μ-distribution for different testproblems”. Source: http://people.ee.ethz.ch/~sop/download/supplementary/testproblems/, Jul 2016.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Graduação
Av. Ipiranga, 6681 - Prédio 1 - 3º. andar
Porto Alegre - RS - Brasil
Fone: (51) 3320-3500 - Fax: (51) 3339-1564
E-mail: prograd@pucrs.br
Site: www.pucrs.br