UPC **CTTC**

Parallel optimization algorithms for High Performance Computing. Application to thermal systems.

Centre Tecnològic de Transferència de Calor
Departament de Màquines i Motors Tèrmics
Universitat Politècnica de Catalunya

Imanol Aizpurua Udabe
Doctoral Thesis

Parallel optimization algorithms for High Performance Computing. Application to thermal systems.

Imanol Aizpurua Udabe

TESI DOCTORAL

presentada al

Departament de Màquines i Motors Tèrmics
E.S.E.I.A.A.T.
Universitat Politècnica de Catalunya

per a l'obtenció del grau de

Doctor per la Universitat Politècnica de Catalunya

Terrassa, February 28, 2017

Parallel optimization algorithms for High Performance Computing. Application to thermal systems.

Imanol Aizpurua Udabe

Director de la tesi

Dr. Assensi Oliva Llena

Tribunal Qualificador

Dr. Antonio Pascau Benito
Universidad de Zaragoza

Dr. Cristóbal Cortés Gracia
Universidad de Zaragoza

Dr. Francesc Xavier Trias Miquel
Universitat Politècnica de Catalunya

To my family.

Wayfarer, the only way
is your footprints and no other.
Wayfarer, there is no way.
Make your way by going farther.
By going farther, make your way
till looking back at where you've wandered,
you look back on that path you may
not set foot on from now onward.
Wayfarer, there is no way;
only wake-trails on the waters.

Antonio Machado
(translated by A. Z. Foreman)

A mi familia.

Caminante, son tus huellas
el camino y nada más;
caminante, no hay camino,
se hace camino al andar.
Al andar se hace camino
y al volver la vista atrás
se ve la senda que nunca
se ha de volver a pisar.
Caminante no hay camino
sino estelas en la mar...

Antonio Machado

Aknowledgments

The end of the Doctoral Thesis is a good chance to look back and observe the timeline of events that took place during these almost five years. As it usually happens, and even if the research activity has been highly enriching, the traces left by collateral experiences lived during this period are the most valuable treasure I will take with me and have decisively contributed to my personal growth. I can't help smiling every time I remember that innocent boy who arrived at Barcelona and the new dimensions of life he came to discover. In this sense, I am happy to notice that time did not go by uselessly. I would like to thank the people with whom I had the occasion to share important experiences, especially Estela, for the time spent together.

I am also grateful for the love and guidance received from my parents during all my life. I think that my effort to join my father's sense of discipline and search for excellence with my mother's tact and practical point of view has been worth it.

Next, I would like to thank the teachers I had in my early years at kindergarten and school for their job. Education is a long process and, as it happens with research, the greatest dedication and the most remarkable results do not necessarily arise at the same time but are part of the same single process.

The completion of this Doctoral Thesis would not have been possible without the collaboration of other researchers and staff from the Heat and Mass Transfer Technological Center (CTTC). I would like to thank especially Prof. Assensi Oliva for believing in me and for giving me the opportunity to join the CTTC, as well as the support and coordination received during the development of the Thesis. His determination and ability to overcome any bureaucratic barrier are laudable and have been really useful. Other members of the group to whom I am grateful are Oriol Lehmkuhl, for his guidance at the early development stage of this Thesis; Joan López, for the time spent philosophizing with me about object oriented programming and parallelization; Jorge Chiva, for checking the accuracy of the Thesis regarding computational topics and

his suggestions; and the computer scientists Octavi Pavón and Ramiro Alba, for their support every time I was about to desperate struggling with my computer.

In the personal aspect, I appreciate the conversations and points of view shared with Daniel Martínez. I feel that we have been sources of mutual inspiration from time to time. Finally, I would like to thank my former roommates and members of the Latin section of the MALET project, Santiago Torras and Nicolás Ablanque, for their good mood and team work. It has been a pleasure to share this time with you.

Without further delay, here is my little contribution to the development of parallel optimization algorithms for High Performance Computing.

Imanol Aizpurua Udabe

Agradecimientos

El final de la Tesis Doctoral es una buena ocasión para volver la vista atrás y observar los acontecimientos de estos últimos casi cinco años. Como suele ocurrir, y aunque la actividad investigadora haya sido muy enriquecedora, el tesoro más valioso que me llevaré es el poso dejado por experiencias colaterales vividas durante este periodo y que han contribuido decisivamente a mi crecimiento personal. No puedo evitar sonreír cada vez que me acuerdo de aquel chico inocente que llegó a Barcelona y de las nuevas dimensiones de la vida que vino a descubrir. En este sentido, me alegra ver que el tiempo no ha pasado en balde. Me gustaría agradecer a la gente con la que he tenido ocasión de compartir experiencias importantes, especialmente a Estela, el tiempo que hemos pasado juntos.

Estoy agradecido también a mis padres por el cariño y la orientación recibidos durante mi vida. Creo que mi esfuerzo por aunar el sentido de la disciplina y la búsqueda de la excelencia de mi padre con el tacto y punto de vista prácticos de mi madre ha merecido la pena.

A continuación querría felicitar por su trabajo a los profesores que tuve en el colegio desde muy temprana edad. La educación es un proceso largo y, como sucede con la investigación, la dedicación más intensa y los resultados más notorios no tienen lugar necesariamente al mismo tiempo aunque sean parte de un único proceso.

La culminación de esta Tesis Doctoral no habría sido posible sin la colaboración de otros investigadores y personal del Centro Tecnológico de Transferencia de Calor (CTTC). En primer lugar, agradezco al Prof. Assensi Oliva haber creído en mí y haberme dado la oportunidad de incorporarme al CTTC, así como el soporte y coordinación recibidos durante el transcurso de la Tesis. Su determinación y habilidad para superar cualquier traba burocrática son loables y han sido de gran utilidad. Otros miembros del grupo a los que estoy agradecido son Oriol Lehmkuhl, por haberme guiado en la fase inicial del desarrollo de esta Tesis; Joan López, por el tiempo dedicado a filosofar

conmigo sobre la programación orientada a objetos y la paralelización; Jorge Chiva, por supervisar la exactitud de la Tesis en lo concerniente al campo de la computación y por sus sugerencias; y los informáticos Octavi Pavón y Ramiro Alba, por su ayuda cada vez que estuve a punto de desesperarme mientras me peleaba con el ordenador.

En el aspecto personal, aprecio las conversaciones y puntos de vista compartidos con Daniel Martínez. Creo que de tanto en tanto hemos sido fuentes de inspiración mutua. Por último, me gustaría agradecer a mis antiguos compañeros de sala y miembros de la sección latina del proyecto MALET, Santiago Torras y Nicolás Ablanque, su buen humor y trabajo en equipo. Ha sido un placer compartir mi tiempo con vosotros.

Sin más preámbulos, aquí está mi pequeña contribución al desarrollo de algoritmos de optimización paralelos para la computación de alto rendimiento.

Imanol Aizpurua Udabe

Contents

Abstract

The need of optimization is present in every field of engineering. Moreover, applications requiring a multidisciplinary approach in order to make a step forward are increasing. This leads to the need of solving complex optimization problems that exceed the capacity of human brain or intuition. A standard way of proceeding is to use evolutionary algorithms, among which genetic algorithms hold a prominent place. These are characterized by their robustness and versatility, as well as their high computational cost and low convergence speed. Such drawbacks are usually tackled by hybridizing them with local search methods, e.g. gradient methods, in order to obtain significant speed up. The use of multiple levels of fidelity of the objective functions is also a common practice.

Many optimization packages are available under free software licenses and are representative of the current state of the art in optimization technology: single-objective and multi-objective search techniques, global and local search methods, plenty of genetic operators, mixed integer optimization capacity, fitness landscape analysis techniques, parallelization strategies, etc. However, the ability of optimization algorithms to adapt to massively parallel computers reaching satisfactory efficiency levels is still an open issue. Even packages suited for multilevel parallelism encounter difficulties when dealing with objective functions involving long and variable simulation times. This variability is common in Computational Fluid Dynamics and Heat Transfer (CFD & HT), nonlinear mechanics, etc. and is nowadays a dominant concern for large scale applications.

Current research in improving the performance of evolutionary algorithms is mainly focused on developing new search algorithms. Nevertheless, there is a vast knowledge of sequential well-performing algorithmic suitable for being implemented in parallel computers. The gap to be covered is efficient parallelization. Moreover, advances in the research of both new search algorithms and efficient parallelization are additive, so that the enhancement of current state of the art optimization software can be accelerated if

both fronts are tackled simultaneously.

The motivation of this Doctoral Thesis is to make a step forward towards the successful integration of Optimization and High Performance Computing capabilities, which has the potential to boost technological development by providing better designs, shortening product development times and minimizing the required resources. A generic mathematical optimization tool has been developed for this aim, applicable in any field of science and engineering. Nevertheless, being this research activity hosted by the Heat and Mass Transfer Technological Center (CTTC), a special focus has been put on the application of the library to the fields of expertise of the Center: Computational Fluid Dynamics and Heat Transfer (CFD & HT), multi-physics simulation, etc.

This document is structured in four chapters. A thorough state of the art study is conducted in the first chapter with the aim of obtaining a global scope of the mathematical optimization techniques available to date, as well as their most remarkable virtues and shortcomings. After classifying optimization problems according to their principal characteristics, such as the number of objective functions (single-objective vs multi-objective) or the nature of the optimization variables (real vs discrete), an insight of constraint handling techniques, surrogate-based optimization and hybrid optimization methods is provided. The most widespread global search algorithm is then introduced, namely the genetic algorithm, followed by a description of the main concepts and shortcomings of the standard parallelization strategies available for such population-based optimization methods. Finally, an overview of the most common test suites for optimization algorithms is given together with some remarks related to random number generators.

The second chapter explains how the implementation of the new optimization library Optimus has been carried out based on the research on optimization theory conducted in the first chapter. The first step has been the definition of the design requirements of the library, including both general code development patterns and specific requirements for the optimization tool. A state of the art study of currently available optimization libraries is then included, taking into consideration open-source and proprietary software. Once the development strategy of the new library is fixed, the main features of Optimus are introduced. Finally, several validation tests are performed in order to demonstrate the suitability of the new library for solving benchmark mathematical optimization tests and real-world CFD & HT optimization problems.

The third chapter contains the main contribution of this Doctoral Thesis. It is started with an approach to the computational load balancing problem detected in the first chapter when the state of the art study on the parallelization of genetic

and other population-based optimization algorithms was carried out. The core of the problem is that processors are often unable to finish the evaluation of their queue of individuals simultaneously and need to be synchronized before the next batch of individuals is created. Consequently, the computational load imbalance is translated into idle time in some processors. This fact was identified as the key point causing the degradation of the optimization algorithm's scalability (i.e. parallel efficiency) in case the average makespan of the batch of individuals is greater than the average time required by the optimizer for performing inter-processor communications. According to the methodology defined for developing load balancing algorithms, the load balancing problem is split into two sub-problems: the estimation of the time required to process each individual, and the subsequent resolution of a combinatorial task scheduling problem in order to map tasks to processors with a certain precedence relation and in the most efficient manner with the aim of reducing the evaluation makespan of each batch of individuals. Several load balancing algorithms are proposed and exhaustively tested by means of 3 theoretical case studies using the genetic algorithm. Being the latter the most widespread optimization heuristic, the impact of the research is expected to be maximized. Note that the proposed algorithms and the reached conclusions are extendable to any other population-based optimization method that needs to synchronize all processors after the evaluation of each batch of individuals. Since time estimation techniques have not been studied in detail due to lack of time, the availability of perfect individuals' evaluation time estimations has been assumed. However, a first implementation of time estimation techniques together with some preliminary tests is included towards the end of the chapter. Finally, a real-world engineering application that consists on optimizing the refrigeration system of a power electronic device is presented as an illustrative example in which the use of the proposed load balancing algorithms is able to reduce the simulation time required by the optimization tool.

The fourth chapter gathers the main conclusions of the conducted research and outlines the next steps to be followed in this approach towards the integration of Optimization Techniques and High Performance Computing.

1

State of the art of optimization algorithms

Abstract. A thorough state of the art study is conducted in this first chapter with the aim of obtaining a global scope of the mathematical optimization techniques available to date, as well as of their most remarkable virtues and shortcomings. Being the application field of interest that of Computational Fluid Dynamics & Heat Transfer (CFD & HT), which is characterized by the simulation of computationally expensive non-linear equation systems, special emphasis is put on the parallelizability of the optimization algorithms.

1.1 Introduction

Every human activity is characterized by the search of Best: sport, social relations, work... There are plenty of things in our daily life which we try to optimize, such as the time spent going from home to our work place, the time spent cooking, the budget for our holidays, our personal appearance, etc.

However, perfection is a quite philosophical concept. Although it guides human actions, reality cannot be understood without constraints. In real life, we never have the option of satisfying all our wishes at the same time: getting closer to perfection in some aspect pushes us to imperfection in some other. Hence, finding a compromise solution among all real options becomes our only feasible goal. In this sense, optimization could be defined as the art of making the most of something rather than the search for perfection.

Once the labyrinth formed by the objective (or objectives) and the constraints is defined, the next question arises: how to get to the goal? We know our starting point, we know our destination and we know the rules of what is feasible and infeasible. Assuming an explorer's role, we lack the map and the compass that can guide us on the way. This orientation technique is what Optimization Theory takes care of. Optimization is the technique that brings explorers to their destination through the shortest way, avoiding that they ran out of resources before reaching their goal. Using a more rigorous language, it can be said that Optimization Theory encompasses the quantitative study of optima and methods for finding them [1].

Given life's and world's complexity, optimization techniques require a thorough study. The search for the right way to the goal is not easy, and the explorer will encounter plenty of "distractions" that may mislead him in his mission. He might think his search has finished, just because he has found some interesting place during the journey. Usually the goal is only one, and the explorer's orientation technique should be sophisticated enough to let him know the improvability of his findings, until he reaches the final Goal.

Leaving the explorer's analogy aside, but following the same reasoning, let's move to the field of engineering and applied mathematics. In any design process, we have a device, mathematical process or experiment which we want to optimize according to some criteria. The optimality measure that will allow us to say if a solution is better than another one is the **objective function**, also called **cost function** or **fitness function**. As previously said, there may be one or multiple objective functions, and the engineer will want to maximize or minimize them. The output of the objective function is dependent on its inputs, which correspond to the characteristics of the

device, experiment, etc. These inputs will be called **optimization variables** from now on. Moreover, the optimization problem may be constrained, meaning this that each optimization variable cannot be modified independently from others.

As stated in [2], several categories may be distinguished in optimization:

- Single-variable / multi-variable: Optimization becomes increasingly difficult as the number of dimensions (number of optimization variables) increases.
- Static / dynamic: Dynamic optimization means that the output is a function of time, while static means that the output is independent of time.
- Constrained / unconstrained: Constrained optimization incorporates variable equalities and inequalities into the objective function, whereas unconstrained optimization allows the variables to take any value. A constrained variable may often be converted into an unconstrained variable through a mathematical transformation. Consider the simple constrained example of minimizing $f(x)$ over the interval $-1 \leq x \leq 1$. The variable x may be converted into an unconstrained variable u by letting $x = sin(u)$ for any value of u.
- Discrete / continuous: Discrete variables (combinatorial optimization) have only a finite number of possible values, whereas continuous variables have an infinite number of possible values.
- Single-objective / multi-objective: Single-objective optimizations seek to improve a unique objective, whereas multi-objective optimizations search for a compromise optimal solution for a set of objectives.

Since usually a fitness surface has many peaks, valleys and ridges (see Fig. 1.1), a difficulty with optimization is to determine if a given optimum (minimum or maximum) is the global optimum or a local optimum (suboptimal value). In this sense, some optimization methods have explorative nature, whereas other methods have exploitative nature. **Exploration** refers to the ability of keeping a global view of the solutions space, and allows distinguishing the global or local nature of an optimum. However, a drawback of explorative methods is their low performance when converging to the optimum in the neighborhood of a known solution. On the other hand, **exploitation** refers to the ability of converging fast and accurately to a local optimum starting from some solution in the neighborhood of that optimum. Nevertheless, the limitation of exploitative algorithms is their incapacity to know if the solution found is a local or global optimum.

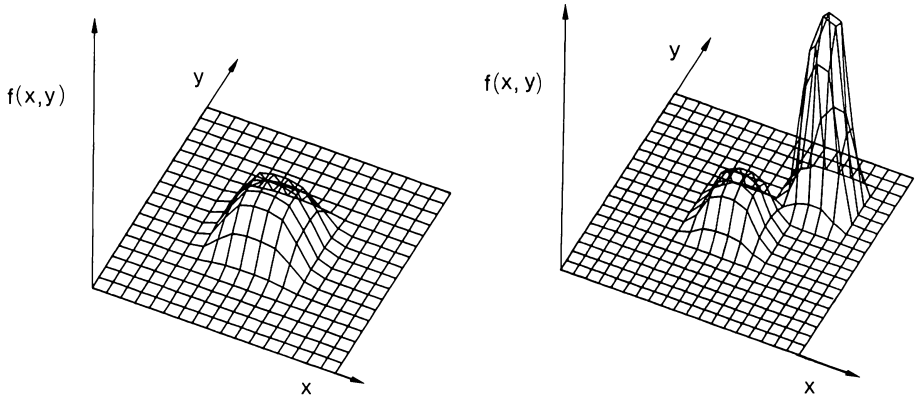


Figure 1.1: Examples of fitness surfaces extracted from [3]. Left: A surface with a single local and global optimum. Right: A surface with two local optima, but a single global optimum.

Many optimization algorithms have been proposed so far, both with explorative and exploitative characteristics. Unfortunately, it is impossible to affirm that one single algorithm will outperform all others for all classes of optimization problems. This assertion is known as the "no free lunch theory" [4]. Since the degree of success of any optimization process depends on the structure of the optimization algorithm and the topology of the objective function, a balance between several algorithms is to be found in order to build a robust and well performing optimization method. Such mixed methods are known as "hybrid optimization algorithms" and combine individual constituent optimization algorithms in a sequential or parallel manner so that the resulting method can utilize the advantages of each constituent algorithm. They usually incorporate at least one explorative method and one exploitative method. All different constituent methods are joined together by a switching algorithm.

This state of the art study is mainly focused on the resolution of continuous single-objective optimization problems, although a rough overview on discrete and multi-objective optimization is also provided in order to help the reader understand the whole context encompassed by mathematical optimization. Most common optimization algorithms are mentioned and classified next, including explorative algorithms (usually heuristic algorithms) and exploitative algorithms (usually deterministic algorithms).

1.2 Single-objective vs. Multi-objective optimization

1.2.1 Single-objective optimization

A rigorous definition of a single-objective optimization problem (either continuous or discrete) is the following according to [5]:

Definition 1.1 General Single-Objective Optimization Problem: *A general single-objective optimization problem is defined as minimizing (or maximizing)* $f(\boldsymbol{x})$ *subject to* $g_i(\boldsymbol{x}) \leq 0$, $i = \{1,...,m\}$, *and* $h_j(\boldsymbol{x}) = 0$, $j = \{1,...,p\}$ $\boldsymbol{x} \in \Omega$. *A solution minimizes (or maximizes) the scalar* $f(\boldsymbol{x})$ *where* $\boldsymbol{x}$ *is a n-dimensional decision variable vector* $\boldsymbol{x} = (x_1,...,x_n)$ *from some universe* Ω.

Plenty of algorithms have been proposed for the resolution of single-objective optimization problems. The following classification offers an overview of the most reputed search techniques:

- **Enumerative algorithms:** This is the simplest search strategy, which consists on evaluating each possible solution within some defined finite search space. The strategy is clearly inefficient if not infeasible. The method belonging to this category is called ***exhaustive search*** or ***parametric study*** [6]. It is the most utilized approach, despite its brute force nature. Checking an extremely large but finite solution space is required, composed by a certain number of combinations of different variable values. A drawback of this method, in addition to the high computational cost, is that the global optimum may be missed by insufficient sampling. This method is only practical for a small number of variables in a limited search space. A possible refinement to the exhaustive search could be first carrying out a coarse sampling of the fitness function and then progressively narrowing the search to promising regions. More sophisticated variations, such as the ***exhaustive interpolation search*** [6] have also been proposed.
- **Deterministic algorithms:** These methods try to optimize the objective function by starting from an initial set of variable values. They are fast in general, but tend to get stuck in local optima (they have exploitative nature). A possible classification of the historically most successful methods is presented next. Bear in mind that the same method may belong to more than one category.
  - ***Calculus-based algorithms*** (also called ***analytical optimization algo-***

rithms or ***gradient methods***): These algorithms are able to quickly find a single optimum, provided that every function is continuous and analytical derivatives can be computed. Derivatives could also be taken numerically, but it would result in more function evaluations and a loss of accuracy. As an example, in the case of a single variable optimization problem an extremum is found by setting the first derivative of the objective function to zero and solving for the variable value. If the second derivative is greater than zero, the extremum is a minimum, and conversely, if the second derivative is less than zero, the extremum is a maximum. In the case of having an objective function of two or more variables, the gradient is calculated and set to zero. Unfortunately, real-life problems are seldom as nice as calculus problems and do not necessarily fulfill the restrictive requirements of continuity and derivative existence. Search spaces may be discontinuous, not smooth and even noisy, as the less calculus-friendly function depicted in Fig. 1.2. Some methods belonging to this category are the ***steepest descent method*** [6], the ***conjugate gradient method*** [6] and ***Newton's method*** [6]. The ***simplex method*** [7], which has been the most popular method for solving linear programming problems, is also included in this category.

- ***Nelder-Mead downhill simplex method*** [8]: This local method does not require the calculation of derivatives. Later, the simplex method was extended giving rise to the ***complex method*** [9], which stands for constrained simplex method.
- ***Successive line minimization methods***: All algorithms belonging to this family follow the same scheme (see Fig. 1.3). They begin at some random point on the cost surface, choose a direction to move and then move in that direction until the cost function begins to increase. Next the procedure is repeated in another direction. The difference between algorithms lies on how the search direction at step n is found. Three common approaches have been the ***coordinate search method*** [10], the ***steepest descent algorithm*** [6] and the ***conjugate gradient method*** [6]. The steepest descent algorithm was originated by Cauchy in 1847 and has been extremely popular. It starts at an arbitrary point on the cost surface and minimizes along the direction of the gradient. But this method, in general, converges slowly for non-quadratic functions. The conjugate gradient method tries to improve the convergence rate of the steepest descent algorithm by choosing the directions of descent that reach the minimum value of the objective function faster. More efficient

techniques have been developed since then, most of them involving some form of the ***Newton's method***. In all these methods the Hessian is needed, but it is not usually known. If it is known, the Newton's method is being used. If it is not known but approximated, a ***quasi-Newton method*** is being used. There are two main quasi-Newton methods that construct a sequence of approximations to the Hessian: the ***Davidon-Fletcher-Powell (DFP)*** algorithm [11] and the ***Broyden-Fletcher-Goldfarb-Shanno (BFGS)*** algorithm [12–15]. These algorithms have the advantage of being fast, but are only able to find an optimum close to the starting point. Quadratic programming assumes that the cost function is quadratic (variables are squared) and the constraints linear. This technique is based on Lagrange multipliers and requires derivatives or approximations to derivatives. Two known powerful methods are the ***recursive quadratic programming*** [16] and the ***sequential quadratic programming*** [17].

- ***Other methods***: Branch & bound [18], greedy method [5], etc.

- **Stochastic methods:** These methods use probabilistic transition rules in the optimum's search. They are slower than deterministic algorithms, but have greater success at finding the global optimum. These algorithms do not require taking objective function derivatives and can thus deal with discrete variables and non-continuous objective functions.

  - ***Random search/walk:*** A random search is the simplest stochastic search strategy, as it simply evaluates a given number of randomly selected solutions. A random walk is very similar, except that the next solution evaluated is randomly selected using the last evaluated solution as a starting point [3]. The drawback of these two methods is their low efficiency.
  - ***Tabu search [19]:*** It is a meta-strategy developed to avoid getting stuck on local optima. It keeps a record of both visited solutions and the "paths" which reached them in different "memories". This information restricts the choice of solutions to evaluate next.
  - ***Simulated annealing [6, 20]:*** This method is based on the thermodynamics of the cooling of a material from a liquid to a solid phase. If a liquid material (e.g. liquid metal) is slowly cooled and left for a sufficiently long time close to the phase change temperature, a perfect crystal will be created, which has the lowest internal energy state. On the other hand, if the liquid

material is not left for a sufficient long time close to the phase change temperature, or if the cooling process is not sufficiently slow, the final crystal will have several defects and a high internal energy state. The gradient-based methods move in directions that successively lower the objective function value when minimizing the value of a certain function or in directions that successively raise the objective function value in the process of finding the maximum value of a certain function. The simulated annealing method can move in any direction at any point in the optimization process, thus escaping from possible local minimum or local maximum values. We can say that gradient-based methods "cool down too fast", going rapidly to an optimum location which, in most cases, is not the global, but a local one.

– ***Natural optimization methods:*** These methods mimic nature in the search of optimum and are based in animal behavior, evolution, social relations, etc. The two most widespread algorithms based on animal behavior are the ***particle swarm optimization*** [21] and the ***ant colony optimization*** [22], but other methods such as the ***predator-prey*** algorithm [2] are also used. However, the family of stochastic algorithms that has risen to fame in the last decades is the one of ***evolutionary algorithms*** [10], abbreviated as EAs. From their point of view, the organisms of today's world should be imagined as being the results of many iterations of a gigantic optimization algorithm, being survivability the objective function that is wanted to be maximized. Natural selection would be the mechanism according to which individuals evolve. According to [23], the main groups of algorithms belonging to this family are ***genetic algorithms*** [24], ***evolution strategies*** [25] (to which the ***differential evolution*** [26] belongs), ***evolutionary programming*** [27], ***genetic programming*** [28] and ***classifier systems*** [25]. Nevertheless, the advancement or optimization of the human race cannot be totally attributed to genetics and evolution. Human interactions, societal behaviors, and other factors play major roles in the optimization process as well. Based on the fact that social interactions allow for faster adaptation and improvement than genetics and evolution, a new group of optimization methods called ***cultural algorithms*** [29] has appeared in recent years.

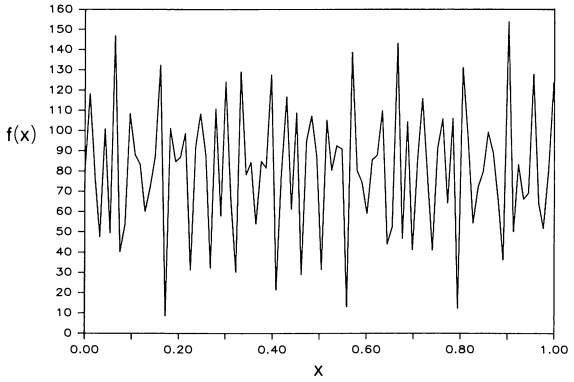


Figure 1.2: Example of a noisy, not calculus-friendly fitness function extracted from [3].

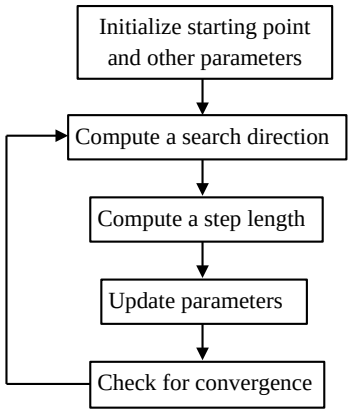


Figure 1.3: Flowchart of a standard line search algorithm.

1.2.2 Multi-objective optimization

The Multi-objective Optimization Problem (MOP) is defined according to [30] as the problem of finding "a vector of decision variables which satisfies constraints and optimizes a vector function whose elements represent the objective functions. These functions form a mathematical description of performance criteria which are usually in conflict with each other. Hence, the term "optimize" means finding such a solution which would give the values of all the objective functions acceptable to the decision

maker". Thus, the goal is to optimize (maximize or minimize) k objective functions simultaneously. A global MOP problem can be formally defined as in [31]:

Definition 1.2 General Multi-Objective Optimization Problem (MOP): *A general MOP is defined as minimizing (or maximizing)* $F(\boldsymbol{x}) = (f_1(\boldsymbol{x}), ..., f_k(\boldsymbol{x}))$ *subject to* $g_i(\boldsymbol{x}) \leq 0$, $i = \{1, ..., m\}$, *and* $h_j(\boldsymbol{x}) = 0$, $j = \{1, ..., p\}$ $\boldsymbol{x} \in \Omega$. *An MOP solution minimizes (or maximizes) the components of a vector* $F(\boldsymbol{x})$ *where* $\boldsymbol{x}$ *is a n-dimensional decision variable vector* $\boldsymbol{x} = (x_1, ..., x_n)$ *from some universe* Ω. *It is noted that* $g_i(\boldsymbol{x}) \leq 0$ *and* $h_j(\boldsymbol{x}) = 0$ *represent constraints that must be fulfilled while minimizing (or maximizing)* $F(\boldsymbol{x})$ *and* Ω *contains all possible* $\boldsymbol{x}$ *that can be used to satisfy an evaluation of* $F(\boldsymbol{x})$.

The main conceptual difference between the single-objective and multi-objective optimizations is the difficulty when comparing two possible solutions in the latter case. A single-objective optimization seeks to maximize or minimize a unique objective, so two objective values can be easily compared in order to decide which solution is the best. On the other hand, and following the same logic, when more than one objective is present a solution may beat another one according to some objectives, but not according to others. If such a situation arises, how should a decision be made?

In order to answer this question, three important concepts of multi-objective optimization are introduced next: fitness assignment, diversity preservation and elitism.

Fitness assignment

Fitness assignment schemes may be classified into four different categories [32]:

- **Scalar approaches**, where the MOP is reduced to a single-objective optimization problem, for instance by means of a weighted-sum aggregation.
- **Criterion-based approaches**, where each objective function is treated separately. In VEGA (Vector Evaluated Genetic Algorithm) [33], for instance, a parallel selection is performed where solutions are discerned according to their values on a single objective function, independently from the others.
- **Dominance-based approaches**, where a dominance relation is used to classify solutions. The main techniques are i) dominance-rank techniques, which compute the number of population items that dominate a given solution, ii) dominance-count techniques, where the fitness value of a solution corresponds to the number of individuals that are dominated by this solution, and iii) dominance-depth

strategies, which classify a set of solutions into different classes or fronts. Hence, a solution that belongs to a class does not dominate another one from the same class.

- **Indicator-based approaches**, where the fitness values are computed by comparing individuals on the basis of a quality indicator I. The chosen indicator represents the overall goal of the search process. Examples of indicator-based EAs are IBEA (Indicator-Based EA) [34] or SMS-EMOA (S-Metric Selection Evolutionary Multi-objective Optimization Algorithm) [35].

The scalar and criterion-based approaches are the most simplistic strategies, being dominance-based and indicator-based approaches usually preferred. These two techniques provide a set of optimal solutions, not just a single solution. But no matter the selected strategy, some criterion is needed in order to select a set of solutions rather than another one. Dominance-based approaches commonly use the so called Pareto-dominance criterion (see [36] for more details on Pareto's Optimality Theory), although some new techniques appeared recently based on other dominance operators such as ε-dominance [37] or g-dominance [38].

Some definitions related to Pareto's Optimality Theory are introduced next, assuming a multi-objective minimization problem:

Definition 1.3 Pareto Optimality [31]: *A solution $\boldsymbol{x} \in \Omega$ is said to be Pareto Optimal with respect to (w.r.t.) Ω if and only if (iff) there is no $\boldsymbol{x'} \in \Omega$ for which $\boldsymbol{v} = F(\boldsymbol{x'}) = (f_1(\boldsymbol{x'}), ..., f_k(\boldsymbol{x'}))$ dominates $\boldsymbol{u} = F(\boldsymbol{x}) = (f_1(\boldsymbol{x}), ..., f_k(\boldsymbol{x}))$. The phrase Pareto Optimal is taken to mean with respect to the entire decision variable space unless otherwise specified.*

In other words, **x*** is Pareto optimal if there exists no feasible vector **x** which would decrease some criterion without causing a simultaneous increase in at least one other criterion.

Definition 1.4 Pareto Dominance [31]: *A vector $\boldsymbol{u} = (u_1, ..., u_k)$ is said to dominate another vector $\boldsymbol{v} = (v_1, ..., v_k)$ (denoted by $\boldsymbol{u} \preceq \boldsymbol{v}$) if and only if $\boldsymbol{u}$ is partially less than $\boldsymbol{v}$, i.e., $\forall i \in \{1, ..., k\}$, $u_i \leq v_i \wedge \exists i \in \{1, ..., k\} : u_i < v_i$.*

Definition 1.5 Pareto Optimal Set [31]: *For a given MOP, $F(\boldsymbol{x})$, the Pareto Optimal Set, P^*, is defined as:*

$$P^* := \{ \boldsymbol{x} \in \Omega \mid \neg\exists\, \boldsymbol{x'} \in \Omega \;\; F(\boldsymbol{x'}) \preceq F(\boldsymbol{x}) \}$$

Summarizing, Pareto optimal solutions are those solutions within the decision space whose corresponding objective vector components cannot be all simultaneously improved. When plotted in the objective space, the non-dominated vectors are collectively known as the Pareto front. Although single-objective optimization problems may have a unique optimal solution, MOPs usually have a possibly uncountable set of solutions on a Pareto front. Each solution associated with a point on the Pareto front is a vector whose components represent trade-offs in the decision space. This is the reason why defining an MOP's global optimum is not a trivial task as the "best" compromise solution depends on the preferences of the (human) decision maker. Thus, the Pareto front represents the "best" solutions available and allows the definition of an MOP's global optimum.

Diversity preservation

Approximating the Pareto optimal set is not only a question of convergence. For a good characterization of the true optimal Pareto front, the final approximation must also be well spread over the objective space. Therefore, a diversity preservation mechanism is usually integrated into the algorithm to uniformly distribute the population over the trade-off surface. Popular examples of Evolutionary Multi-objective Optimization (EMO) diversity assignment techniques are sharing and crowding [32]. Sharing consists on estimating the distribution density of a solution using a so-called sharing function that is related to the sum of distances to its neighborhood solutions. Crowding allows maintaining diversity without specifying any parameter. It consists in estimating the density of solutions surrounding a particular point of the objective space.

Elitism

Elitism [32] consists on maintaining an external set (archive) that allows storing either all or a subset of non-dominated solutions found during the search process. This secondary population aims at preventing the loss of these solutions during the stochastic optimization process, and is continuously updated with new potential non-dominated solutions. Even if an archive is usually used as an external storage only, archive members can also be integrated during the selection phase of an EMO algorithm, leading to elitist EMOs.

Keeping in mind the concepts introduced so far, and being a Pareto front the solution provided by the optimization algorithm to the engineer, the main goals of a multi-objective optimization algorithm may be established as follows [32]:

- Convergence, i.e. the distance of the resulting non-dominated set to the true Pareto front should be minimized.
- Diversity, i.e. the Pareto front must be well characterized in its whole length, avoiding a high concentration of solutions in one area and lacking solutions in another area.
- Elitism, i.e. non-dominated points in the objective space and associated solution points in the decision space must be preserved.

Several ways have been proposed in order to achieve the goals mentioned above. The following classification of multi-objective optimization algorithms, very similar to the one proposed for single-objective methods, gathers the most widely spread algorithms to date:

- **Enumerative algorithms:** This is the family of the inefficient brute-force algorithms, such as the ***exhaustive search***.
- **Deterministic algorithms:** The range of available algorithms for multi-objective optimization is more limited than for single-objective optimization, or the use of such methods is at least less spread [5]. ***Depth-first search (hill-climbing)*** [39] and ***gradient-based search methods*** could be mentioned as the most usual strategies. The latter are only applicable to MOPs consisting of continuous variables, being the derivatives obtained based upon a specific direction from selected non-dominated points in the known Pareto front with the aim of moving a point towards the true Pareto front. Note however that derivatives tend to be quite noisy in such situations.
- **Stochastic algorithms:** Natural optimization methods (mainly ***evolutionary algorithms*** [5]) have remarkably succeeded in solving multi-objective optimization problems. A key advantage has been that the population-based nature of EAs allows the generation of several elements of the Pareto optimal set in a single run. This field is now called Evolutionary Multi-objective Optimization (EMO), which refers to the use of evolutionary algorithms of any sort. The most notorious algorithms currently available are the following: ***MOGA*** [40], ***NPGA*** [41], ***VEGA*** [33], ***NSGA*** [42], ***PAES*** [43], ***NSGA-II*** [44], ***SPEA*** [45], ***SPEA2*** [46], ***PESA*** [47], ***ε-MOEA*** [37, 48] and ***IBEA*** [34]. They use techniques going from a simple linear aggregating function to the most popular Multi-Objective Evolutionary

Algorithms (MOEAs) based on Pareto ranking. Other natural optimization algorithms that are used for multi-objective optimization are the ***multi-objective particle swarm optimization (MOPSO)***, ***cultural algorithms***, ***differential evolution***, ***predator-prey algorithm*** and ***ant colony optimization***. Some additional strategies that do not belong to natural optimization methods are also used, e.g. ***simulated annealing*** [6, 20] and ***tabu search*** [19], but there seems to be a clear preference for the use of evolutionary algorithms.

A brief overview of some stochastic algorithms mentioned above is provided hereafter (based on [49]):

- **Pareto ranking (MOGA):** Fonseca and Fleming [40] proposed a variation of Goldberg's fitness assignment where a solution's rank corresponds to the number of solutions in the current population by which it is dominated.
- **Pareto sharing:** A fitness assignment like the previous one tends to produce premature convergence, what does not guarantee a uniformly sampled final Pareto approximation set. To avoid that, Fonseca and Fleming [40] modified the strategy above by implementing fitness sharing in the objective space to distribute the population over the Pareto-optimal region.
- **Non-dominated Sorting Genetic Algorithm (NSGA):** Srinivas and Deb [42] introduced this algorithm which classifies the solutions into several classes (or fronts). A solution that belongs to a class does not dominate another one from the same class. Logically, the best fitness value is assigned to solutions of the first class, because they are closest to the true Pareto-optimal front of the problem. Diversity is preserved by means of a fitness sharing procedure.
- **NSGA-II:** This is a modified version of NSGA introduced by Deb et al. [44]. The algorithm is computationally more efficient, uses elitism and keeps diversity by means of a crowding technique.
- **Strength Pareto Evolutionary Algorithm (SPEA):** This elitist algorithm was proposed by Zitzler and Thiele [45]. It maintains an external population (an archive) that stores a fixed number of non-dominated solutions found during the optimization process in order to define the fitness of a solution based on these archive members.

- **SPEA2:** It is an improved version of SPEA, introduced by Zitzler et al. [46]. In comparison to its predecessor, SPEA2 includes an improved fitness assignment technique, a density estimation technique and an archive truncation method.

- **Indicator-Based Evolutionary Algorithm (IBEA):** Introduced by Zitzler and Künzli [34], it has the characteristic to compute fitness values by comparing individuals on the basis of an arbitrary binary quality indicator I (also called binary performance metric). Thereby, no particular diversification mechanism is necessary. The indicator, determined according to the decision maker preferences, denotes the overall goal of the optimization process. Two binary quality indicators commonly used are the additive ε-indicator [50] and the I_{HD}-indicator [50] that is based on the hypervolume concept [45] (see [50] for an overview about quality indicators). IBEA is a good illustration of the new EMO trend dealing with indicator-based search that started to become popular in recent years.

Once the goals for multi-objective optimization methods have been established, some performance assessment method is to be developed in order to know which algorithm works best. The existing performance metrics can be classified into three classes [51]:

- Convergence metrics: They evaluate how far the known Pareto front is with respect to the true Pareto front.

- Diversity metrics: They evaluate how scattered the final population of the Pareto front is.

- Metrics for both convergence and diversity: They evaluate both the distance to the true Pareto front and the dispersion of final solutions.

Various quality indicators have been proposed in the literature for evaluating the performance of multi-objective search methods [32, 50, 52–54]: entropy, contribution, generational distance, spacing, coverage of two sets, coverage difference, S-metric, D-metric, R-metrics, hypervolume metric (both in unary and binary form), additive and multiplicative ε-indicators, etc. However, none of the metrics can be considered the best, being usually necessary to use more than one metric to evaluate the performance of the multi-objective evolutionary algorithms. The reader is referred to [50] for a general review.

1.3 Constraint handling

Realistic engineering problems are always multidisciplinary. Consequently, constraints of both equality and inequality type are very likely to appear. A set of design variables that does not violate any constraints is said to be feasible, while design variables that violate one or more constraints are infeasible. Every general purpose optimization algorithm must be able to handle the existence of these equality and inequality constraints. Based on [55], the following are the most common ways and their use makes sense depending on the nature of the algorithm (gradient based, non-gradient based, etc.):

- **Restauration method** (also called **feasible search):** Designs that violate constraints are automatically restored to feasibility via the minimization of the active global constraint functions.
- **Rosen's projection method:** Provides search directions which guide the descent direction tangent to active constraint boundaries.
- **Penalty method:** The fitness of the designs that violate constraints is artificially worsened, with the aim of forcing the optimization algorithm to abandon that search region of the solutions space.
- **Random design generation:** When an infeasible design is detected, it is discarded and new random designs are generated within a (for instance, Gaussian-shaped) probability density cloud about a desirable and feasible design until a new design is reached.

The constraint handling techniques must be carefully chosen and implemented, because they have a direct effect on the convergence of the optimization algorithm.

1.4 Surrogate-based optimization

One of the main concerns when running an optimization process is the computational cost. The evaluation of real-life objective functions can be very expensive, since it might involve solving non-linear systems, big meshes, etc. Taking into account that the optimization could need hundreds or thousands of objective function evaluations, the need for computational resources may seem scary. Therefore, for problems where objective function evaluations are already expensive and where the number of design variables is large (thus requiring many objective function evaluations), the only economically

viable approach to optimization is to use an inexpensive and as accurate as possible surrogate model (a metamodel) instead of the actual high fidelity analysis method. Such surrogate models are known as response surfaces. They are very useful at the early stages of optimization, although progressively more complete physical models should be used as the global optimization process starts converging. Surrogate models are fitted through the available (often small) set of high fidelity values of the objective function. Once the response surface is created using an appropriate analytic formulation, it is very easy and fast to search such a surface for its optimum given a set of values of design variables supporting such a response surface. Some basic concepts related to the response surface generation methodology are presented in [6].

From the viewpoint of kernel interpolation/approximation techniques, many response surface methods are based on linear and non-linear regression and other variants of the least square technique. This group of mesh-free methods has been successfully applied to many practical, but difficult problems in engineering that are to be solved by the traditional mesh-based methods. The commercial optimization software IOSO (Indirect Optimization Based Upon Self-Organization) [56], a software known for its extraordinary speed and robustness, partly owes its success to the appropriate use of response surfaces.

Due to the existence of several response surface techniques, their performance is to be evaluated according to some criterion. Here are some key aspects worth taking into account proposed by [57]:

- Accuracy: The capability of predicting the system response over the design space of interest.
- Robustness: The capability of achieving good accuracy for different problem types and sample sizes.
- Efficiency: The computational effort required for constructing the metamodel and for predicting the response for a set of new points by metamodels.
- Transparency: The capability of illustrating explicit relationships between input variables and responses.
- Conceptual simplicity: Ease of implementation. Simple methods should require minimum user input and be easily adapted to each problem.

A crucial aspect for the proper construction of a response surface by means of any algorithm is the location of the training points. If we are given freedom to choose

the locations of the support points of a multi-dimensional response surface, a typical approach is to use Design of Experiments (DoE) for this purpose. For high dimensional problems, strategies such as the Latin Hypercube Sampling [58] or a variety of random number generators (e.g. the Sobol quasi-random sequences of numbers [59]) are used. However, when we do not have freedom to choose the number and the locations of the support points, all existing methods for generating response surfaces have serious problems with accuracy and robustness. This is mainly because arbitrary data sets provide inadequate uniformity of coverage of space of the design variables and clustering of the support points that leads to oscillations of the response surfaces.

The most common multidimensional response surface fitting algorithms and their hybrids have been described in [60]: **polynomial regression** [61], **kriging** [62], **radial basis functions** [63], **neural networks** [64] and **self-organizing algorithms** [65]. Hybrid methods may also be constructed in order to overcome the shortfalls of single methods. The proposed hybrid methods are the **Fittest Polynomial Radial Basis Function (FP-RBF)**, **Kriging Approximation with Fittest Polynomial Radial Basis Function (KRG-FP-RBF)**, **Hybrid Self-Organizing Model With RBF** and the **Genetic Algorithm Based Wavelet Neural Network (HYBWNN)**. The article evaluates the performance of these algorithms on data sets containing either a scarce, small, medium or large number of points.

1.5 Hybrid optimization methods

The "no free lunch theory" [4] has been introduced in a previous section. This theory says it is impossible to affirm that one single algorithm will outperform all others for all classes of optimization problems. Therefore, the usual way to proceed is the creation of hybrid methods, also called metaoptimization or hyperheuristics. Two are the key aspects to be discussed regarding hybridization:

- **Constitutive algorithms of a hybrid method may be combined sequentially, in parallel or in a mixed sequential/parallel way [60].** In sequential hybridization, a control algorithm performs automatic switching among the constituent algorithms at each stage during the optimization when the rate of convergence becomes unsatisfactory, the process tends towards a local optimum, or some other undesirable aspect of the iterative process appears [55]. In parallel hybridization, constitutive optimization algorithms run in parallel and contribute a portion of each new generation's population. The portion that each search contributes to the new generation is dependent on the success of the algorithm to

provide past useful solutions to the search. Finally, a sequential/parallel method is a mix of the two other hybridization strategies. Nevertheless, the resulting hybrid method is expected to be more robust and converge faster than its individual constituent optimization algorithms no matter the selected hybridization technique. In the context of this Doctoral Thesis, only sequential hybridization has been considered.

- **Combining both deterministic and stochastic methods allows achieving a right balance between exploration and exploitation.** Deterministic methods are in general computationally faster (they require fewer objective function evaluations) than stochastic methods, although they can converge to a local minimum or maximum, instead of the global one. On the other hand, stochastic algorithms can ideally converge to a global maximum or minimum, although they are computationally slower than the deterministic ones. Indeed, stochastic algorithms can require thousands of evaluations of the objective functions and, in some cases, become non-practical. This is why stochastic methods are usually employed to find the region where the global optimum is located and deterministic methods to get the exact optimal point.

The simplest sequential hybrid optimization algorithm is composed by a global (stochastic) search method and a local (deterministic) search method. Regarding the stochastic method, evolutionary algorithms hold a prominent place since their appearance in 1975, being well suited for both continuous and discrete optimization. Genetic algorithms are the most extended method in the evolutionary algorithms' family. Regarding the deterministic method, it differs depending on the continuous or discrete nature of the optimization problem.

Historically, single-objective continuous optimization problems have been thoroughly studied and gradient methods are the most widely used local search methods. However, there is less consensus in the case of continuous multi-objective optimization problems [5]. A survey of deterministic methods for continuous single-objective optimization is carried out in [6]. According to this study, **Newton's method** converges more rapidly than the **conjugate gradient method**, but it has the drawback of long calculation times for the Hessian matrix coefficients. Therefore, it is preferred to approximate the Hessian based only on first order derivatives and avoiding the calculation of second order derivatives. This is done by means of the so called quasi-Newton methods, which have a slower convergence rate than the Newton's method, but are overall computationally faster. Anyway, quasi-Newton methods converge more rapidly than the conjugate gradient

method. The two most popular quasi-Newton methods are the **Davidon-Fletcher-Powell (DFP)** and the **Broyden-Fletcher-Goldfarb-Shanno (BFGS)** methods. The latter is a variation of DFP, being less sensitive to the choice of the search step size. If the possibility of transforming the original optimization problem is considered, **Sequential Quadratic Programming (SQP)** offers the best performance, being the successive quadratic problems solved by means of the BFGS algorithm.

As stated in the introduction of this chapter, continuous single-objective optimization is the application field of most interest for this Doctoral Thesis. This is the reason why the information provided on discrete or combinatorial optimization, either single-objective or multi-objective, is scarcer. However, the reader is referred to [5] to get a deeper insight on the most widespread discrete local search methods, such as the greedy algorithm, depth-first search, hill-climbing, branch and bound, etc. A review on hybrid single-objective optimization algorithms is available in [6, 55], whereas [5, 60] provide an overview on hybrid multi-objective optimization algorithms.

1.6 The genetic algorithm

1.6.1 Basic concepts

The genetic algorithm (GA) is an optimization and search technique based on the principles of genetics and natural selection, where a population of solutions is iteratively improved under specified stochastic operators and selection rules to a state that maximizes the fitness (i.e. minimizes the cost function). The genetic algorithm is the most widely used evolutionary algorithm, was first developed by John Holland [24] around 1975 and later popularized by David E. Goldberg [3]. The mathematical foundations of genetic algorithms were established in Goldberg's book in 1989, and it is still a reference for current research in the field.

Genetic algorithms owe their reputation to the fact that they [2]:

- are suitable for continuous and discrete variables.
- do not require derivative information.
- search simultaneously in a wide range of the cost surface.
- can deal with a large number of variables.
- are well suited for parallel computing.

- are able to optimize variables with extremely complex cost surfaces not getting stuck in local optima.
- provide a list of optimum variable sets, not just a single solution.
- work with numerically generated data, experimental data, or analytical functions.

Genetic algorithms are based on the concept of biological evolution by means of natural selection. This is why vocabulary taken from biology's field is employed when referring to certain aspects of the algorithm. Some basic definitions are the following [23]:

- **Genotype:** The code devised to represent the parameters of the problem in the form of a string.
- **Gene:** The encoded version of a variable of the problem being solved, i.e. each characteristic that the optimization algorithm is able to modify in order to search for better solutions to the optimization problem.
- **Chromosome:** One encoded string of optimization variables, i.e. a set of values given to the genes (one value per gene). A chromosome represents a potential solution to the optimization problem.
- **Individual:** One chromosome with its associated objective value(s), i.e. a set of values for the optimization variables together with a quality measure.
- **Fitness:** Real value indicating the quality of an individual as a solution to the problem.
- **Population:** A set of individuals with their associated statistics (fitness average, etc.).

A schematic representation of some of the above concepts is shown in Fig. 1.4.

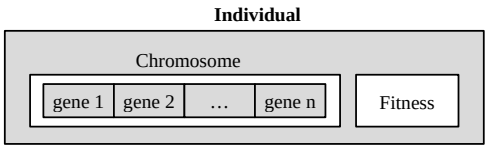


Figure 1.4: Schematic representation of an individual belonging to a genetic algorithm.

1.6.2 Structure of the algorithm

A flow chart of the typical structure of a genetic algorithm is represented in Fig. 1.5. A brief explanation of each step is provided hereafter [2, 32]:

Definition of the optimization problem: cost/fitness function, optimization variables, constraints
This is a strongly problem dependent step, common to any optimization algorithm, and consists on the correct formulation of the optimization problem. First the original decision variables are defined. Then a representation of these variables is to be chosen. What is meant by representation? Original variables may not be the best choice to be used as input variables (genes) to the optimization algorithm. Therefore, any mathematical transformation (encoding) is allowed with the aim of improving the solvability of the solutions' space, as long as the process is reversible and the original decision variables may be recovered (decoding). The explanation of this phenomenon is that representation affects the way solutions are initialized and variation operators applied. Various encodings have been used, such as binary variables, real-coded vectors, permutations, etc.

It must be reminded that GAs originated with a binary representation of the variables, and the binary GA fits nicely when the nature of optimization variables is discrete. However, the variables may be continuous and full machine precision required. In this case, it is more logical to represent the variables by floating-point numbers, giving rise to continuous GAs.

The problem's definition is finished by specifying the objective function(s) and the equality and inequality constraints.

Selection of GA parameters
Finding optimized GA parameters (also called metaoptimization) is a tough and also a strongly problem dependent task. The success and performance of a GA depend on the configuration of several aspects of the optimization algorithm, such as the type and rate of crossover, type and rate of mutation, elitist strategy, etc. In the case of multi-objective optimization the customization possibilities are even wider, including strategies for fitness assignment, diversity preservation and archive management. This topic is not studied in the scope of the present Doctoral Thesis, but the reader is referred to [2] for a deeper insight. Adaptive parameter setting techniques have also been proposed, giving rise to Adaptive Genetic Algorithms (AGAs) [66].

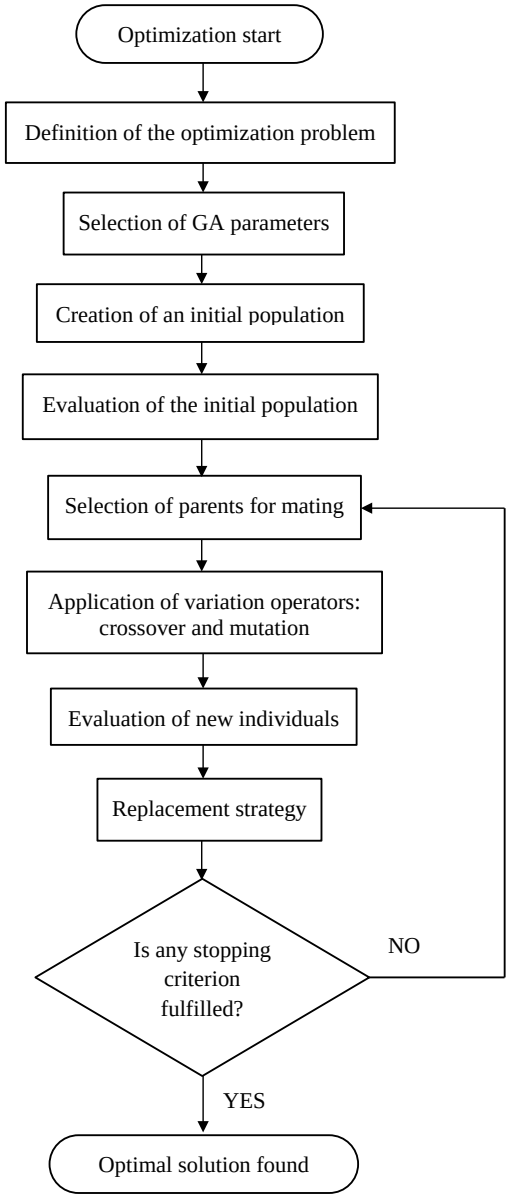


Figure 1.5: Flow chart of a genetic algorithm.

Creation of an initial population
The goal is to create a well-diversified initial population, covering the whole decision space, in order to prevent premature convergence to find the global optimum. Since no information on the solution is a priori available in the most general case, the initial population is randomly generated.

Evaluation of the initial population
The chromosomes of the initial population are evaluated and a fitness value is assigned to them.

Selection of parents for mating
This selection step consists of choosing some individuals that will be used to generate the offspring population. In general, the better (fitter) an individual is, the higher is its chance of being selected (survival of the fittest law, according to natural selection). Common selection strategies are deterministic or stochastic tournament, roulette-wheel selection, random selection, etc. The parents are chosen from the last population obtained, but in the case of multi-objective problems, individuals from the archive may also be included in the parent selection process if required by an elitist scheme.

Application of variation operators: crossover and mutation
The purpose of variation operators is to modify the individuals belonging to the previous generation in order to move them in the search space. These problem-dependent operators are stochastic and of two types: crossover and mutation.

Crossover (or recombination) operators are mostly binary, and sometimes n-ary. The most common form involves two parents that produce two offspring, but any number of parents could be selected and any number of offspring produced. Thus, offspring contain portions of the genetic code of parents. Many different approaches have been tried for crossing over, either in discrete GAs (one-point crossover, two-point crossover, etc.) or continuous GAs (uniform crossover, blending methods, heuristic crossover, etc.) [67, 68]. Since the best performing operator is not usually known a priory, many codes combine various methods to use the strengths of each.

Mutation operators are unary random operators acting on a single individual, altering a certain percentage of genes of each offspring. Increasing the number of mutations increases the algorithm's freedom to search outside the current region of variables' space. Again several approaches exist for mutation, either in discrete GAs (bit flip mutation,

uniform mutation, etc.) or continuous GAs (Gaussian mutation, uniform mutation, etc.).

As a consequence of the application of variation operators, an offspring population is created. The reader is referred to [2] for a deeper insight on variation operators.

Evaluation of new individuals
Each offspring generated in the previous step is evaluated, i.e. all objective values related to the chromosome are calculated.

Replacement strategy
Survivors are selected from both the current and the offspring populations in some arbitrary way. In generational replacement, the offspring population systematically replaces the parent one. In an elitist strategy, the N best solutions are selected from both populations, where N stands for the appropriate population size.

Convergence check
Since the GA is an iterative method, stopping criteria are to be defined and checked after each generation. The most common stopping criteria are the following: a given maximum number of iterations (generations), a given maximum number of objective function evaluations, a given run time, a target design, solution not improving in a certain number of subsequent generations, etc.

If any stopping criterion is fulfilled, the optimization algorithm has finished. If not, parents are again selected for mating and the population evolves towards a new generation.

End of the optimization algorithm: solution found
The best individual of the last generation is considered to be the solution to the optimization problem.

1.7 Parallelization of genetic and other population-based optimization algorithms

1.7.1 General concepts

As explained in the previous section, genetic algorithms maintain a population of potential solutions that evolves for a certain number of generations. For nontrivial problems this process might require high computational resources (due to large search

times, for example). Hence, the total time of the optimization process is to be reduced by means of parallelization in order to make the use of genetic algorithms viable.

Genetic algorithms are naturally prone to parallelism since the operations on the individuals are relatively independent from each other. According to [23], parallelization techniques can be divided into software and hardware parallelization. In the case of genetic algorithms, when both classes of parallelization techniques are applied together an exceptional characteristic arises: the behavior of the parallel algorithm is better than the sum of the separate behaviors of its component sub-algorithms, i.e. the new algorithm is not just the parallel version of a sequential algorithm intended to provide speed gains.

Software parallelization is accomplished by using a structured population, either in the form of a set of islands or a diffusion grid, and often leads to superior numerical performance even when the algorithms run on a single processor. Comparing with natural evolution, software parallelization is based on the fact that species form a large population distributed in a certain number of semi-isolated breeding subgroups. The local selection and reproduction rules allow the species evolve locally, and diversity is enhanced by migrations of individuals among the interconnected subgroups. Different search techniques may be utilized in each subgroup.

Hardware parallelization is an additional way of speeding up the execution of the algorithm and consists on running a sequential algorithm in several processors. The results obtained in the parallel execution are the same as in the sequential execution, but a certain speedup is achieved in run time. What acceleration could be expected? According to [69], it is possible to have super-linear speedup for certain problems and parameterizations by using hardware parallelization exclusively, both in homogeneous and in heterogeneous parallel machines. The concept of super-linear speedup is understood as the fact that using m processors leads to an algorithm that runs more than m times faster than the sequential version. This assertion is compliant with [70], where Donaldson et al. showed that there is no theoretical upper limit for the speedup in heterogeneous systems.

Another important concept related to parallelization is heterogeneity, where two different kinds may be distinguished again.

Heterogeneity at software level refers to the use of different search techniques (coding, operators, parameters, etc.) in the population subgroups created by means of software parallelization. If the same search techniques are applied in all subgroups, the algorithm is considered homogeneous at software level.

Heterogeneity at hardware level refers to the existence of processors with dif-

ferent characteristics (architecture, clock rate, etc.) when a genetic algorithm is run on a network of computers or in massively parallel computers. If the characteristics of all processors where the optimization is run are the same, the hardware is considered homogeneous.

Finally the concept of scalability is introduced. **Scalability** measures the ability of a parallel machine and a parallel algorithm to use efficiently a larger number of processors, and depends on both the communication patterns of the algorithm and the infrastructure provided by the machine. Several scalability measures are described in [71].

1.7.2 Main parallelization strategies

According to [23], parallelization strategies of genetic algorithms are commonly divided into 3 types: global parallelization (type 1), coarse grain parallelization (type 2) and fine grain parallelization (type 3). Type 1 corresponds to hardware parallelization. In types 2 and 3, the classification coarse/fine grain relies on the computation/communication ratio. If this ratio is high, the Parallel Genetic Algorithm (PGA) is called a coarse grain algorithm, whereas if low it is called a fine grain PGA. Coarse grain PGAs are the most popular techniques and are also known as distributed or island GAs (dGAs), whereas fine grain PGAs are known as cellular (cGAs), diffusion or massively-parallel GAs. Hybrid algorithms have also been proposed, combining different parallel GAs at two levels in order to enhance the search in some way (see schemes d, e and f in Fig. 1.6). An interesting comparison of various parallel implementations may be found in [72].

Type 1: Global parallelization (master-worker parallelization)
Also called explicit parallelization, this strategy implements hardware parallelization and distributes genetic operations and/or objective functions evaluations among several processors (a fraction of the population is assigned to each of the processors). This type presents a viable choice only for problems with a time-consuming function evaluation. Otherwise the communication overhead is higher than the benefits of the parallel execution.

According to [73], the global parallel GA is called synchronous if it proceeds in the same way as a sequential GA, i.e. stopping and waiting to receive the fitness values for all the population before starting the next generation. This is the most usual strategy. The global parallel GA is called asynchronous when there is no clear division between

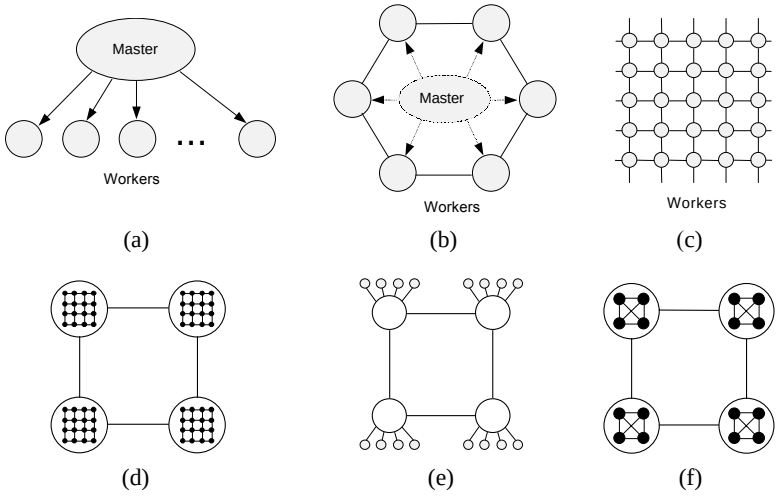


Figure 1.6: Different models of PGA (extracted from [23]): (a) global parallelization, (b) coarse grain parallelization, (c) fine grain parallelization, (d) coarse + fine grain hybrid, (e) coarse grain + global hybrid, and (f) coarse grain + coarse grain hybrid.

generations, i.e. if any worker processor that finishes evaluating an individual returns it to the master and receives another individual. A high level of processor utilization is achieved with this strategy, despite the workers' heterogeneous processor speeds. However, the obtained search results differ from those achieved by the sequential GA.

The global parallelization uses a single population. The main advantage is the simplicity of implementation, because parallelization takes place only at the level of objective function calculation. The disadvantage is that no software parallelization is utilized. A schematic representation is provided in Fig. 1.6 (a).

The method can be implemented efficiently on shared- and distributed-memory computers [73]. On a shared-memory multiprocessor, the population can be stored in shared memory and each processor could read a fraction of the population and write back the evaluation results without any conflicts. On a distributed-memory computer, the population is stored in one processor. This "master" processor is the responsible for sending the individuals to the other processors (the "workers") for evaluation, collecting the results, and applying the genetic operators to produce the next generation. The difference with a shared-memory implementation is that the master has to send and receive messages explicitly.

Type 2: Coarse grain parallelization (island GA, distributed GA)
As it was mentioned in a previous section, this strategy is based on distributing the whole population in a certain number of semi-isolated breeding subgroups called islands. An independent optimization takes place in each island, where a new population of individuals is created from the old one by applying genetic operators such as selection, crossover, mutation and replacement. From time to time some individuals are exchanged among the interconnected subgroups (see Fig. 1.6 (b)).

Migration is the operator that guides this exchange of individuals among islands. The main concepts related to migration, which are the object of study of most publications on the island GA field [74], are mentioned hereafter.

The ***migration gap*** is the number of steps (generations) in every sub-population between two successive exchanges. Migration may take place in every island periodically or by using a given probability of migration to decide in every step whether migration will take place or not. Island GAs are usually synchronous, i.e. the phases of migration and reception of external migrants are embedded in the same portion of the algorithm (migration takes place when all islands achieve a fixed number of generations). However, synchronous migration has the drawback of being slow for some problems [23]. In asynchronous island GAs, migrants are sent whenever it is needed and migrants are accepted whenever they arrive. This behavior is accomplished by implementing a chromosome buffer [74].

The ***migration rate*** is the parameter determining the number of individuals that undergo migration in every exchange. It is not clear which is the best value for the migration rate, but best results have been obtained for low percentages (ranging from 1% to 10% of the population size).

Selection and replacement operators in the migration procedure of each island are commonly the same as the ones used in the independent optimization process taking place in that island.

The ***topology*** is the map of interconnections between islands, being the general tendency to use static topologies that are set before launching the algorithm and remain unchanged [74]. It seems that the ring and hyper-cube are two of the best topologies for most problems, whereas full-connected and centralized topologies have shown problems in their parallelization and scalability due to the tight connectivity. Another approach is the use of dynamic topologies, giving rise to decentralization. In this kind of topology, all populations can be connected to all other populations directly. The main disadvantage of this approach is the complexity of implementation and, moreover, the efficiency and

usefulness of dynamic topologies has not been proved [74].

The performance of distributed GAs is often better than for the sequential GAs. The two main reasons are that subpopulations are run simultaneously using several processors (reducing the whole processing time) and that this kind of search maintains samples of very different promising zones of the search space, increasing the efficacy of the algorithm.

The coarse-grain parallelization has several advantages with respect to fine-grain parallelization. On one hand, it is possible to adapt existing sequential algorithms for being used in each subpopulation (island). On the other hand, the number of available processors does not affect the result of the optimization. Thus, the strategy is suitable in the case of having limited resources. Moreover, it could be adapted more easily than the cellular GA to grid computing [75], where the number and performance of processors may vary during the optimization.

The main disadvantage of a multiple-population GA might be that the critical path of a fine-grained algorithm seems to be shorter [76].

Type 3: Fine grain parallelization (cellular GA, diffusion GA, massively parallel GA)

A cellular GA is implemented at large computer terminals [74] and consists of one spatially distributed population in which overlapping subpopulations execute the same reproductive plan. In every neighborhood the new individual computed after selection (usually by means of a binary tournament), crossover and mutation replaces the current one only if it is better. This process is repeated for all the neighborhoods in the grid of the cellular GA, where there are as many neighborhoods as individuals. The most popular algorithm uses a toroidal grid of individuals (toroidal topology) and defines a NEWS neighborhood (North-East-West-South) in which subpopulations of 5 individuals (4+1) execute the reproductive plan. Thus, "good" individuals are spread over the whole distributed population by means of migrations happening between neighboring subpopulations (4 subpopulations available per individual if a NEWS neighborhood is used). A schematic representation is provided in Fig. 1.6 (c).

This strategy is very well suited for massively parallel computers, because it divides the population into a large number of parts. One individual is usually evaluated in each processor. Thus, the cellular GA is a completely decentralized model in which communications are equally distributed and where each cell has to wait only to a few other cells. Moreover, it is proven in [76] that the critical path of a fine-grained algorithm is shorter than that of a multiple-population GA. This means that if enough

processors were available, massively parallel GAs would need less time to finish their execution, regardless of the population size. Note, however, that considerations such as the communications bandwidth or memory requirements were not included in the mentioned theoretical study.

The main disadvantage of this strategy is that the use of a small number of processors results in degeneration of the whole population, leading the genetic algorithm into a local optimum [74]. Therefore, this model should not be utilized unless a minimum number of processors is available. Another drawback may be that sequential algorithms cannot be adapted to fine-grain parallelization, being necessary the development of completely new algorithms.

Hybrid parallelization models

When two or more GA parallelization methods are combined they form a hierarchy [76], i.e. a hybrid parallelization strategy, and a better performance than with any of the constituent methods alone is achieved. Several distinct combinations may be proposed, as it is shown in Fig. 1.6 (d), (e) and (f). Note that these three models use a coarse-grained algorithm at the upper level, which is usual in hierarchical GAs [73].

A common hybrid strategy consists on combining a coarse-grained GA at the upper level and a global GA (master-worker model) at each of the subpopulations [76]. Thus, independent optimizations using the master-worker strategy are carried out in each island and some individuals migrate between islands from time to time. This approach is useful when objective functions that need a considerable amount of computation time are evaluated.

1.7.3 Shortcomings of standard parallelization techniques

Description of shortcomings

The integration of optimization methodologies (and particularly of genetic algorithms) with computational analyses/simulations has a profound impact on product design. Nowadays the main challenges related to the application of such methodologies arise from high-dimensionality of problems, computationally-expensive analysis/simulations and unknown function properties (i.e. black-box functions). An extensive review on these topics is available in [77]. The consequence is that the computational load of the optimization process may be considerable. This fact makes the development of appropriate parallelization techniques a critical issue in order to benefit from optimization strategies in real-world design problems.

The goal of every parallelization strategy is to maximize CPU usage by minimizing the communication overhead and by balancing the computational load correctly, thus avoiding idleness of allocated processors. In the case of traditional optimization algorithms (and particularly of genetic algorithms), the principal causes of computational load imbalance are listed below.

- ***Inappropriate ratio (no. individuals / no. processor groups)***
  Traditionally, when several individuals may be evaluated simultaneously, each individual is assigned to a group of processors. In standard algorithms, all groups are formed by the same number of processors and remain unchanged during optimization. Thus, the number of individuals assigned to a certain group of processors is calculated as the total number of individuals in the population divided by the number of processor groups. In case the residual of this quotient is not zero, some processor groups will evaluate one more individual than the other groups. Hence, the processor groups with fewer individuals to evaluate will remain idle while the rest of the groups finish their pending evaluations. For this explanation, all individuals are assumed to have homogeneous evaluation times and the number of allocated processor groups to be equal or smaller than the number of individuals in the population.

  Such a problem is reported for example in [78], where the optimum shape design of aerodynamic configurations is studied. The genetic algorithm evaluates an objective function that calls an unstructured grid-based CFD solver. The objective evaluation time represents between 80% and 90% of the total elapsed time. A two-level parallelization strategy is utilized, having a master-worker model at the upper level and a parallel evaluation of the objective function at the lower level. Two equally sized processor groups are responsible of concurrently evaluating two individuals. The evaluation of an odd number of airfoils may cause computational load imbalance, influencing negatively the parallel performance of the algorithm.

- ***Heterogeneous parallel computer systems***
  A computer system is a collection of n-processors interconnected by a communication network, specified by the pairwise latency and the bandwidth between processors. Ideally, computer systems would be homogeneous, i.e. all nodes would have identical architecture, clock rate and latency. Unfortunately, a certain degree of heterogeneity is hardly avoidable even between nodes sharing the same architecture, because additional factors such as the hardware's age, refrigeration, etc. affect their performance. This hardware heterogeneity translates into

non-homogeneous objective evaluation times and the consequent loss of parallel performance in the case of optimization algorithms. In this regard, a general model to define, measure and predict the efficiency of applications running on heterogeneous parallel computer systems is presented in [79].

The effect of the imbalance caused by heterogeneous parallel systems is noticeable, for instance, if the parallel genetic algorithm implemented by Cantú-Paz [80] is used. This algorithm was designed for a homogeneous parallel computer system and distributes the same number of individuals per processor. Therefore, a considerable loss of parallel performance can be expected in heterogeneous environments.

The most extreme situation regarding hardware heterogeneity occurs probably in grid computing. Grid computing consists of a geographically distributed infrastructure gathering computer resources around the world, and has emerged as an effective environment for the execution of parallel applications that require great computing power [75]. Grid computing environments provide an attractive infrastructure for implementing parallel metaheuristics. However, the fact that grid resources are distributed, heterogeneous and non-dedicated makes writing parallel grid-aware applications very challenging, because one has to address the issues of grid resource discovery and selection, grid job preparation, submission, monitoring and termination which differ from one middleware to another.

- ***Heterogeneous objective function evaluation time***
  Heterogeneity in the computational cost of evaluating the objective functions may cause an important load imbalance, provided that the simulation time of the genetic algorithm is dominated by the evaluation time of the objective functions. This phenomenon happens when the evaluation cost of individuals is dependent on the optimization variables, i.e. the input variables of the objective functions, and is not unusual in heat transfer and nonlinear mechanics applications.

  This kind of imbalance is reported for example in [78], a publication already mentioned in this document. As it was explained, two individuals are concurrently evaluated in two processor groups. Computational load imbalance is observed and caused by the fact that the required number of iterations for evaluating the objective function depends on the shape of the airfoil, i.e. depends on its optimization variables.

Current research

Ongoing research regarding the previously introduced shortcomings is summarized hereafter. The main causes of computational load imbalance are listed again, together with some currently available solutions.

- ***Inappropriate ratio (no. individuals / no. processor groups)***
  The standard and simplest solution is to adjust the ratio so that the residual of the quotient becomes zero. This may be achieved by either modifying the number of processor groups or resizing the population managed by the genetic algorithm. Nevertheless, the coupling between these two terms is annoying and might involve some drawback when configuring an optimization algorithm to be used in massively parallel computers.

  As an example, a population with an even number of airfoils was created in [78] once the load imbalance was detected due to the use of an odd number of individuals.

- ***Heterogeneous parallel computer systems***
  Several studies have been found which try to tackle this source of imbalance. Some of them are cited hereafter, as a sample of the solutions proposed up to date.

  Genetic algorithms are discussed from an architectural perspective in [81], offering a general analysis of performance of GAs on multi-core CPUs and on many-core GPUs. As a conclusion, the authors propose the best parallel GA scheme for multi-core, multi-socket multi-core and many-core architectures.

  The use of information of processors' heterogeneity to make the distribution of individuals nonhomogeneous is proposed in [79]. However, the approach is said to be highly undesirable because it requires that the program interacts with the resource management software, which contains the speeds of the processors allocated to the job. The implementation of such a strategy results in a non-portable program, due to the lack of standard interfaces for supplying this information.

  In the context of grid computing, some tools for managing hardware heterogeneity have been recently developed. WoBinGO [75], for instance, is a framework for solving optimization problems over heterogeneous resources, including HPC clusters and Globus-based grids. It uses a master-worker parallelization model, which is easily replaceable by a hierarchical parallel GA with master-worker demes or by a parallel cellular GA.

- ***Heterogeneous objective function evaluation time***
  A very basic (and not advisable) solution to homogenize all objective function evaluation times is to limit their maximum allowed duration, e.g. by establishing a maximum number of iterations to their solver (this method is used for instance in [78]). Nonetheless, it must be borne in mind that this alternative affects the obtained results and is not suitable for general use.

- ***Solutions to any kind of heterogeneity (hardware- or software-based)***
  Although the source of hardware- and software-based heterogeneity is different, the consequence is common: both affect the time needed to evaluate each individual. This is the reason why some proposed solutions are valid for these two types of heterogeneity and have been summarized in this section.

  A master-worker model with a constant and homogeneous number of individuals assigned to each processor is described in [73]. However, the need of balancing the computational load among processors using a dynamic scheduling algorithm like guided self-scheduling is mentioned.

  The Adaptive Parallel Genetic Algorithm is proposed in [79]. This algorithm automatically changes the number of individuals to be evaluated on each processor depending on the evaluation time of each individual. The algorithm uses a server–client blocking message architecture, in which the server node is responsible for the distribution of work and the evolution of the genetic algorithm. The client nodes evaluate the fitness function for the individuals and return their values to the server node (see Fig. 1.7). Communication between processors is necessary for distributing and/or balancing the evaluation of a population over the nodes. Using this scheme a fast processor will request more work than a slow one and as a result, the algorithm will send more individuals to the faster processors adapting the algorithm to the heterogeneity of the system. The same will happen in the case of processors that receive individuals with short evaluation times, adapting the algorithm to heterogeneous objective function evaluation times.

  A similar alternative is described in [82], where an asynchronously global parallel genetic algorithm with 3-tournament elimination selection is introduced. The difference between the traditional master-worker GA and this algorithm is in the tasks performed by the master and the workers. In the traditional algorithm workers only evaluate individuals, whereas the master distributes individuals among workers as well as performing all genetic operations. In the new algorithm, the master only initializes the population, whereas workers perform the whole

evolution process including evaluation.

The use of a steady-state algorithm is proposed in [83]. Generational algorithms have to wait for all individuals to be evaluated before going on to the next generation. In the case of having heterogeneity, this implies that some processors might stay idle for a long time, waiting for the slowest evaluations to complete. In the asynchronous steady-state algorithm offspring are sent out for evaluation and inserted back in the population on a first-come first-served basis. Thus, no processor stays idle for a long time. However, if the evaluation costs do depend on the characteristics of the individuals, some region of the search space might be less explored than others. A comparison of the asynchronous steady-state and the generational algorithms is carried out in this paper, both applied to the multi-objective optimization of a Diesel engine.

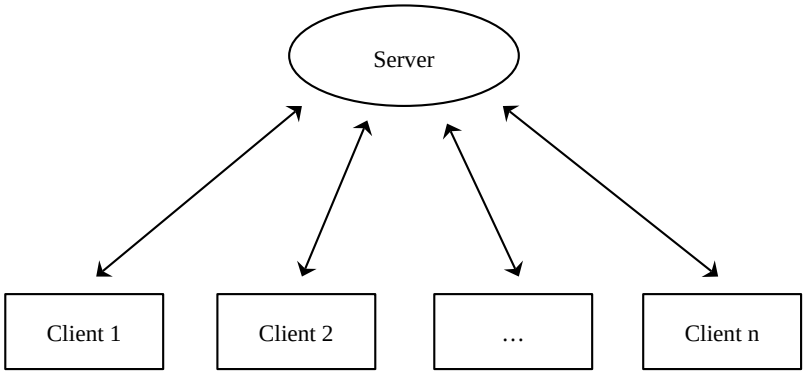


Figure 1.7: Diagram of the Adaptive Parallel Genetic Algorithm.

There are other publications not directly concerned by the load balancing problem previously described, but related to parallelization of genetic and other optimization algorithms. For example, Rivera mentions in [71] the development of high performance parallel genetic algorithms with the feature of choosing among several parallelization alternatives depending on the problem and machine parameters. In [84], Nordin et al. show their concern caused by the low utilization of available computer power in many organizations. They propose to use partially loaded desktop computers (because they are being used for word processing or some other task requiring low computational power) to create a corporative grid for parallel simulation and optimization. A framework

monitoring available computer resources and running distributed simulations is needed for this aim.

Coming back to parallel genetic algorithms, the dominating research area is software parallelization. There are many studies related to topologies, migration, etc. for coarse-grain and fine-grain algorithms. No significant research on master-worker algorithms is apparently being carried out, and scarce research has also been found related to hardware parallelization or heterogeneity in objective function evaluation times.

Future steps

The increasing computational cost of real-world problems is pushing research in the field of supercomputing. We are entering the exascale era, in which millions of cores will be available for each simulation [85]. Such massive parallelism makes necessary the adaptation of existing codes in order to ensure an efficient use of computational resources. Regarding optimization, the inherent parallel nature of evolutionary computation is a promising factor in designing robust and scalable optimization algorithms that adapt to such extreme-scale supercomputers. However, up to now only a handful of papers have been published to solve extremely large-scale optimization problems with extreme-scale parallel evolutionary algorithms implemented on supercomputers. Therefore, evolutionary computation is expected to be an active research area in the near future.

The best publication found by the author related to optimization in massively parallel computers is [86]. Since this document has been taken as a reference for establishing the objectives of this Doctoral Thesis, an extensive summary is included in the following paragraphs with the aim of easing the understanding of the upcoming chapters to the reader.

The opportunities for exploiting parallelism in optimization may be categorized into four main levels:

- ***Concurrent optimizations:*** Coarse-grained parallelism is realized through the concurrent execution of multiple optimizations, e.g. giving rise to the island model.
- ***Concurrent evaluation of individuals within each optimization:*** Coarse-grained parallelism is realized through the concurrent evaluation of multiple individuals within each optimization.
- ***Concurrent analyses within each individual's evaluation:*** Coarse-grained parallelism is exploited when multiple separable simulations are performed as part of evaluating the objective function(s) and constraints.

- ***Multiple processors for each analysis:*** Fine-grained parallelism is exploited when parallel analysis codes are available. The simulation code is responsible for internally distributing work among the simulation processors.

Exploiting a single type of parallelism involves performance limitations that prevent effective scaling with the thousands of processors available in massively parallel (MP) supercomputers. Coarse-grained parallelism is suitable for MP, since it requires very little inter-processor communication. However, it is not usual to have enough separable computations on each optimization to utilize all available processors. Fine-grained parallelism, on the other hand, involves much more communication among processors and care must be taken to avoid the case of inefficient machine utilization. Thus, the way of maximizing the overall efficiency consists on exploiting multiple levels of parallelism.

The difficult task of distributing the computational work among the various parallelization levels is carried out by task scheduling algorithms. Three approaches are listed in [86]:

- ***Self-scheduling:*** The master processor manages a single processing queue and maintains a prescribed number of jobs active on each group of workers. Once a group of workers has completed a job and returned its results, the master assigns the next job to this group. Thus, the workers themselves determine the schedule through their job completion speed. Heterogeneous processor speeds and/or job lengths are naturally handled, provided there are sufficient instances scheduled to balance the variation.

- ***Static scheduling:*** The schedule is statically determined at start-up. If the schedule is good, this approach will have superior performance. However, heterogeneity, when not known a priori, can very quickly degrade performance since there is no mechanism to adapt.

- ***Distributed scheduling:*** In this approach each group or workers maintains a separate queue of pending jobs. When one queue is smaller than the other queues, it requests work from other groups (hopefully prior to idleness). In this way, it can adapt to heterogeneous conditions, provided there are sufficient instances to balance the variation. Each partition performs communication between computations, so no processors are dedicated to scheduling. However, this approach involves relatively complicated logic and additional communication for queue status and job migration, and its performance is not always superior since a partition can become idle if other groups of workers are locked in computation. This logic

can be somewhat simplified if a separate thread can be created for communication and migration of jobs.

Regarding the processor partitioning models, the "dedicated master" approach and the "peer partition" approach are mentioned. In the dedicated master partitioning, a processor is dedicated exclusively to scheduling operations. In the peer partition approach, the loss of a processor to scheduling is avoided. This strategy is desirable since it utilizes all processors for computation. However, it requires either the use of sophisticated mechanisms for distributed scheduling or a problem for which static scheduling of concurrent work performs well.

Some computational experiments were carried out in [86] in order to discover the most efficient partitioning scheme. Under homogeneous conditions, the highest efficiencies for the self-scheduling and static scheduling strategies were achieved when coarse-grain parallelism was prioritized, i.e. when the number of simulations being run simultaneously was maximized. Nevertheless, experiments were also carried out with a stochastic case adding a 10% variation to the simulation duration using an exponential distribution. The results showed a considerable negative impact at the higher parallelism levels, since there are fewer opportunities to balance heterogeneity in job length when coarse-grain parallelism is favored. The conclusion of these experiments is that when heterogeneous simulation times are present, a compromise between coarse-grain and fine-grain parallelism is to be found in order to make a good use of computational resources.

1.8 Test suites for optimization algorithms

The existence of many optimization algorithms, each of them configurable with several parameters, makes the ability of evaluating and comparing algorithms' performance necessary. Nevertheless, and as stated in a previous section, it is impossible to affirm that one single algorithm will outperform all others for all classes of optimization problems ("no free lunch theory" [4]). So how should such a performance evaluation be carried out? Answering this question led researchers to the creation of various test suites with the aim of checking the virtues and weaknesses of each algorithm under different conditions.

Standard test suites are composed by several mathematical functions which try to mislead optimization algorithms in the search of the global optimum. However, the fact that an optimization algorithm performs well in a certain test suite guarantees neither

its success when solving a complex real world problem nor its adequacy for solving optimization problems belonging to a specific field of study.

As a general advice, it is suggested to assess the robustness of optimization algorithms by means of generic mathematical test suites, but also adding tests with specific characteristics from the target problem domain in order to evaluate more accurately the algorithm's performance in the real world problems it will have to face.

The characteristics of single-objective and multi-objective optimization algorithms that should be addressed by a generic test suite are listed hereafter [5]:

- Continuous vs. discontinuous vs. discrete
- Differentiable vs. non-differentiable
- Convex vs. concave
- Modality (unimodal, multi-modal)
- Numerical vs. alphanumeric
- Quadratic vs. non-quadratic
- Type of constraints (equalities, inequalities, linear, nonlinear)
- Low vs. high dimensionality (number of optimization variables)
- Deceptive vs. non-deceptive
- Biased vs. unbiased portions of true Pareto Front

A review of most widespread test suites is carried out in [5], including single-objective, multi-objective, continuous and discrete optimization. Popular suites are for instance De Jong's test suite [87] in single-objective, or Levy functions [88] in multi-objective optimization research. Regarding this Doctoral Thesis, two observations must be made. First, the Thesis is focused on continuous optimization and consequently scarce research has been done related to discrete test suites. Second, the objective is the implementation of already known optimization algorithms and no research is carried out in the development of new methods. This means that no exhaustive check of the optimization algorithms is required, just some validation with available benchmarks. Thus, only a few test functions have been used.

1.8.1 Single-objective optimization tests

In the case of single-objective optimization, the following four tests have been selected: Rosenbrock, Rastrigin and Griewank functions (taken from the De Jong's test suite [87]) and the Schwefel function (taken from [89]). The four functions are defined hereafter and their graphical representation can be found in Fig. 1.8:

Rosenbrock

Minimize

$$f(\langle x_1,...,x_n\rangle) = \sum_{i=1}^{n-1}(1-x_i)^2 + 100(x_{i+1}-x_i^2)^2 \qquad x_i \in [-2.048, 2.048] \tag{1.1}$$

where the minimum $f(x) = 0$ is located at $x = (1,1,...,1)$.

Rastrigin

Minimize

$$f(\langle x_1,...,x_n\rangle) = 10n + \sum_{i=1}^{n} x_i^2 - 10cos(2\pi x_i) \qquad x_i \in [-5.12, 5.12] \tag{1.2}$$

where the minimum $f(x) = 0$ is located at $x = (0,0,...,0)$.

Griewank

Minimize

$$f(\langle x_1,...,x_n\rangle) = 1 + \frac{1}{4000}\left(\sum_{i=1}^{n} x_i^2\right) - \prod_{i=1}^{n} cos\left(\frac{x_i}{\sqrt{i}}\right) \qquad x_i \in [-600, 600] \tag{1.3}$$

where the minimum $f(x) = 0$ is located at $x = (0,0,...,0)$.

Schwefel

Minimize

$$f(\langle x_1,...,x_n\rangle) = \sum_{i=1}^{n} -x_i \ sin\left(\sqrt{|x_i|}\right) + 418.9829n \qquad x_i \in [-512.03, 511.97] \tag{1.4}$$

where the minimum $f(x) \approx 0$ is located at $x = (420.9687, 420.9687, ...)$.

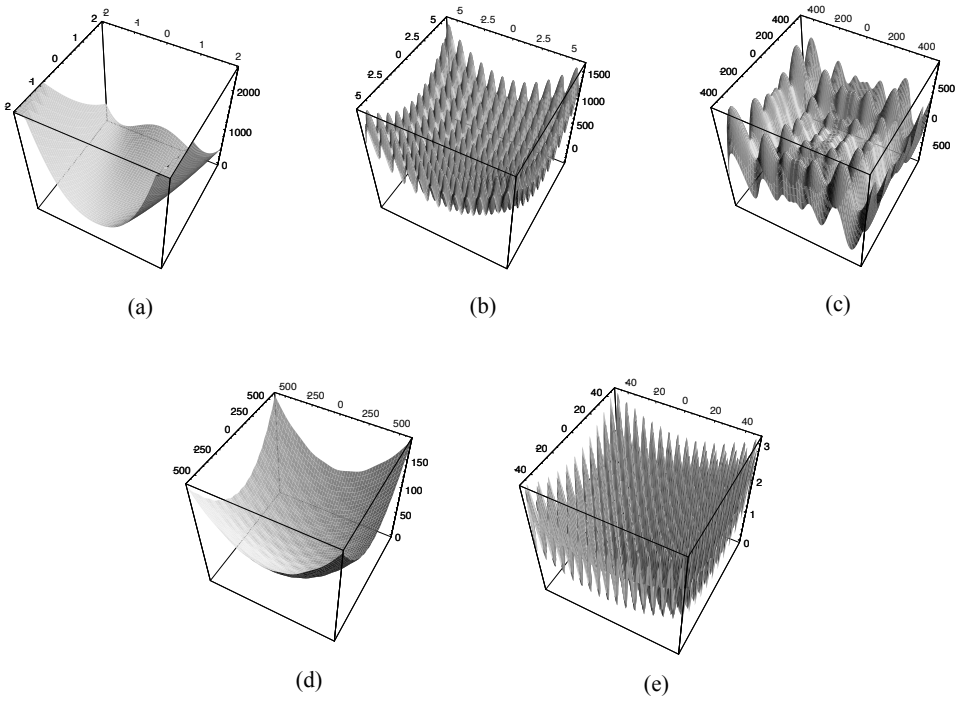


Figure 1.8: Test functions for single-objective optimization (extracted from [90]): (a) Rosenbrock, (b) Rastrigin, (c) Schwefel, (d) Griewank, and (e) Griewank (detail).

1.8.2 Multi-objective optimization tests

In the case of multi-objective optimization, typical test suites are gathered in [5] under the names of MOP, ZDT, DTLZ, OKA, WFG, etc. Two typical tests for unconstrained optimization have been selected, namely MOP4 (see Fig. 1.9) and ZDT6 (see Fig. 1.10):

MOP4 (Kursawe's function)

Minimize the following objective functions

$$f_1(\mathbf{x}) = \sum_{i=1}^{2} \left(-10exp\left(-0.2\sqrt{x_i^2 + x_{i+1}^2} \right) \right) \tag{1.5}$$

$$f_2(\mathbf{x}) = \sum_{i=1}^{3} \left(|x_i|^{0.8} + 5\ sin(x_i)^3 \right) \tag{1.6}$$

where $-5 \leq x_i \leq 5 \quad i = 1,...,3.$

ZDT6 (Zitzler-Deb-Thiele's function N. 6)

Minimize the following objective functions

$$f_1(\mathbf{x}) = 1 - exp(-4x_1)\ sin^6(6\pi x_1) \tag{1.7}$$

$$f_2(\mathbf{x},g) = g(\mathbf{x}) \cdot \left(1 - \left(\frac{f_1(\mathbf{x})}{g(\mathbf{x})} \right)^2 \right) \tag{1.8}$$

where $0 \leq x_i \leq 1 \quad i = 1,...,10$ and

$$g(\mathbf{x}) = 1 + 9 \left(\frac{\sum_{i=2}^{10} x_i}{9} \right)^{0.25} \tag{1.9}$$

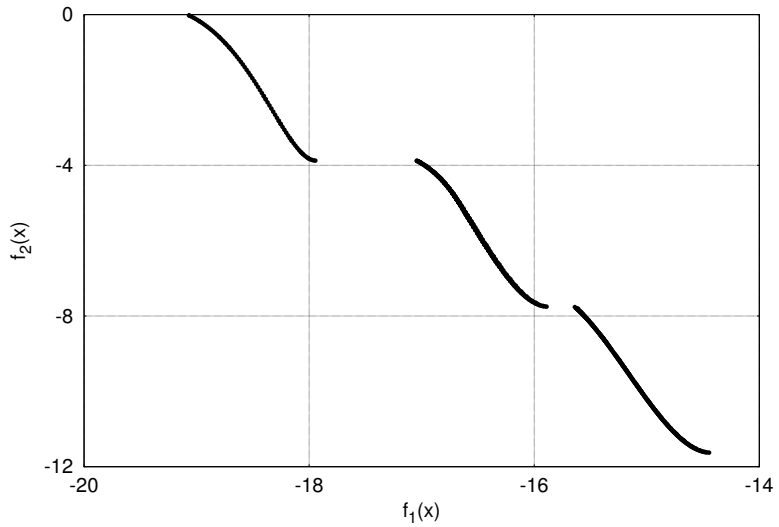


Figure 1.9: True Pareto Front of Kursawe's function (MOP4).

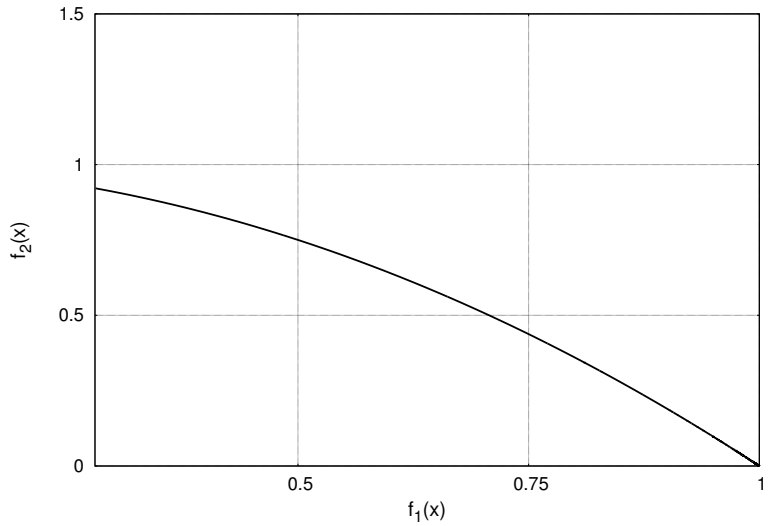


Figure 1.10: True Pareto Front of Zitzler-Deb-Thiele's function N. 6 (ZDT6).

1.9 Random number generators

It has been said that evolutionary algorithms (among others) are stochastic methods, which means that they employ randomness to some degree. This implies that the quality of the produced results also relies on the quality of the selected random number generator. Consequently, it is necessary to carry out a brief state of the art research related to this topic.

A pseudorandom number generator is an algorithm for generating a sequence of numbers whose properties approximate the properties of sequences of random numbers. However, the generated sequence is not truly random, being completely determined by a relatively small set of initial values. These values are called the generator's seed. Distributions of most usual programming languages like C, C++ and Java include pseudorandom number generators. Nevertheless, their quality has been strongly questioned, as in [90], making the search for other alternatives highly advisable.

The renowned mathematical library Intel® Math Kernel Library (MKL) [91] provides various pseudorandom, quasi-random, and non-deterministic random number generators: Wichmann-Hill pseudorandom number generator [92], Mersenne Twister MT19937 pseudorandom number generator [93], SIMD-oriented Fast Mersenne Twister SFMT19937 pseudorandom number generator [94], Sobol quasi-random number generator [59], Niederreiter quasi-random number generator [95], etc. Studies of advantages and deficiencies of random number generators are also available, for instance in [96]. Regarding the consulted bibliography on evolutionary algorithms, references to the choice of an appropriate random number generator have been found in [60], where Sobol's pseudorandom sequence generator [59] is employed, and in [97], where the Mersenne Twister MT19937 [93] is used.

All these considerations have been taken into account in Chapter 2, where a random number generator is selected for implementation.

1.10 Conclusions

The state of the art of optimization algorithms has been studied in this chapter. After a brief introduction to the goals of optimization, the main concepts have been defined and a classification of currently available search techniques for both single-objective and multi-objective optimization has been presented. All those algorithms must be used together with efficient constraint handling methods in order to guarantee the overall efficiency of the search.

The evaluation of objective functions can be computationally expensive in real-life problems, making the cost of the whole optimization process prohibitive. Surrogate-based optimization has been introduced as an effective method of getting a lower fidelity model of the original function which can be simulated in a reasonable time, making possible the application of mathematical optimization techniques to such problems.

Another difficulty faced by the optimization community is the lack of a single search technique which always outperforms all other available techniques. Thus, the best practical solution consists on creating hybrid optimization methods, which are able to select the best performing constituent algorithm at each moment.

Evolutionary algorithms are the most widely used global optimization methods to date, among which genetic algorithms hold a prominent place. The main characteristics of such algorithms have been defined and an extensive review of the current state of the art regarding their parallelization has been carried out.

Common test suites for comparing the performance of optimization algorithms have also been introduced and a selection of some tests has been made.

Finally, the importance of choosing a good random number generator has been remarked in the case of using stochastic optimization algorithms. The most widespread generators have been mentioned, together with some software packages that include their implementation.

Acknowledgments

This work has been financially supported by a FPU Doctoral Grant (Formación de Profesorado Universitario) awarded by the Ministerio de Educación, Cultura y Deporte, Spain (FPU12/06265) and by Termo Fluids S.L. The author thankfully acknowledges these institutions.

References

[1] C. S. Beightler, D. T. Phillips, and D. J. Wilde. *Foundations of optimization*. Prentice Hall Inc., 1979.

[2] R. L. Haupt and S. E. Haupt. *Practical Genetic Algorithms*. John Wiley & Sons, 2004.

[3] D. E. Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley, 1989.

[4] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[5] C. A. Coello Coello, G. B. Lamont, and D. A. Van Veldhuizen. *Evolutionary algorithms for solving multi-objective problems*. Springer Science & Business Media, 2007.

[6] M. J. Colaço and G. S. Dulikravich. A survey of basic deterministic, heuristic and hybrid methods for single-objective optimization and response surface generation. In *METTI IV - Thermal Measurements and Inverse Techniques*, volume 1, Rio de Janeiro, Brazil, 2009.

[7] G. B. Dantzig and R. Cottle. *The basic George B. Dantzig*. Stanford University Press, 2003.

[8] J. A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, 1965.

[9] M. J. Box. A comparison of several current optimization methods, and the use of transformations in constrained problems. *The Computer Journal*, 9(1):67–77, 1966.

[10] H. P. Schwefel. Evolution and optimum seeking. Sixth-generation computer technology series, 1995.

[11] M. J. D. Powell. An efficient method for finding the minimum of a function of several variables without calculating derivatives. *The Computer Journal*, 7(2):155–162, 1964.

[12] C. G. Broyden. A class of methods for solving nonlinear simultaneous equations. *Mathematics of Computation*, 19(92):577–593, 1965.

[13] R. Fletcher. Generalized inverses for nonlinear equations and optimization, Numerical Methods for Non-linear Algebraic Equations. *Gordon & Breach (R. Rabinovitz, Ed.), London, UK*, 1963.

[14] D. Goldfarb and L. Lapidus. Conjugate gradient method for nonlinear programming problems with linear constraints. *Industrial & Engineering Chemistry Fundamentals*, 7(1):142–151, 1968.

[15] D. F. Shanno. An accelerated gradient projection method for linearly constrained nonlinear estimation. *SIAM Journal on Applied Mathematics*, 18(2):322–334, 1970.

[16] D. G. Luenberger and Y. Ye. *Linear and nonlinear programming*, volume 2. Springer, 1984.

[17] J. L. Zhou and A. Tits. User's guide for FFSQP version 3.7: A Fortran code for solving optimization programs, possibly minimax, with general inequality constraints and linear equality constraints, generating feasible iterates. Technical Report SRC-TR-92-107r5, Institute for Systems Research, University of Maryland, 1997.

[18] J. Pearl. *Heuristics: Intelligent search strategies for computer problem solving*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.

[19] F. Glover and M. Laguna. Tabu search. *Kluwer Academic Publishers*, 1997.

[20] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[21] K. E. Parsopoulos and M. N. Vrahatis. Recent approaches to global optimization problems through particle swarm optimization. *Natural Computing*, 1(2-3):235–306, 2002.

[22] M. Dorigo and L. M. Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, 1997.

[23] E. Alba and J. M. Troya. A survey of parallel distributed genetic algorithms. *Complexity*, 4(4):31–52, 1999.

[24] J. H. Holland. *Adaptation in natural and artificial systems*. Ann Arbor: University of Michigan Press, 1975.

[25] T. Bäck, D. B. Fogel, and Z. Michalewicz. *Handbook of evolutionary computation*. IOP Publishing Ltd., Bristol, UK, 1st edition, 1997.

[26] R. Storn and K. Price. Differential evolution: a simple and efficient adaptive scheme for global optimization over continuous spaces. Technical Report TR-95-012, International Computer Science Institute, Berkeley, California, 1995.

[27] L. J. Fogel. *Artificial intelligence through simulated evolution*. John Wiley, New York, 1966.

[28] J. R. Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[29] R. G. Reynolds. An introduction to cultural algorithms. In A. V. Sebald and L. J. Fogel, editors, *Evolutionary Programming - Proceedings of the Third Annual Conference*, pages 131–139, San Diego, CA, USA, 24-26 February 1994. World Scientific Press.

[30] A. Osyczka. Multicriteria optimization for engineering design. *Design Optimization*, 1:193–227, 1985.

[31] D. A. Van Veldhuizen. *Multiobjective evolutionary algorithms: classifications, analyses, and new innovations*. PhD thesis, Wright Patterson AFB, OH, USA, 1999. AAI9928483.

[32] A. Liefooghe, L. Jourdan, and E. G. Talbi. A unified model for evolutionary multiobjective optimization and its implementation in a general purpose software framework: ParadisEO-MOEO. Research Report RR-6906, INRIA, 2009.

[33] J. D. Schaffer. Multiple objective optimization with vector evaluated genetic algorithms. In *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 93–100. L. Erlbaum Associates Inc., 1985.

[34] E. Zitzler and S. Künzli. Indicator-based selection in multiobjective search. In *International Conference on Parallel Problem Solving from Nature (PPSN VIII)*, volume 3242, pages 832–842. Springer-Verlag, 2004.

[35] N. Beume, B. Naujoks, and M. Emmerich. SMS-EMOA: Multiobjective selection based on dominated hypervolume. *European Journal of Operational Research*, 181(3):1653–1669, 2007.

[36] M. Ehrgott. *Multicriteria optimization*. Springer Science & Business Media, 2005.

[37] K. Deb, M. Mohan, and S. Mishra. Evaluating the ϵ-domination based multiobjective evolutionary algorithm for a quick computation of Pareto-optimal solutions. *Evolutionary Computation*, 13(4):501–525, 2005.

[38] J. Molina, L. V. Santana, A. G. Hernández-Díaz, C. A. Coello Coello, and R. Caballero. g-dominance: Reference point based dominance for multiobjective metaheuristics. *European Journal of Operational Research*, 197(2):685–692, 2009.

[39] S. Russell and P. Norvig. *Artificial intelligence: A modern approach*. Prentice-Hall, Upper Saddle River, New Jersey, 1995.

[40] C. M. Fonseca and P. J. Fleming. Genetic algorithms for multiobjective optimization: formulation, discussion and generalization. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, volume 93, pages 416–423, University of Illinois at Urbana-Champaign, San Mateo, California, 1993. Morgan Kaufmann Publishers.

[41] H. Horn, N. Nafpliotis, and D. E. Goldberg. A niched Pareto genetic algorithm for multiobjective optimization. In *Proceedings of the First IEEE Conference on Evolutionary Computation, IEEE World Congress on Computational Intelligence*, volume 1, pages 82–87, Piscataway, New Jersey, 1994.

[42] N. Srinivas and K. Deb. Multiobjective optimization using nondominated sorting in genetic algorithms. *Evolutionary Computation*, 2(3):221–248, 1994.

[43] J. D. Knowles and D. W. Corne. Approximating the nondominated front using the Pareto archived evolution strategy. *Evolutionary Computation*, 8(2):149–172, 2000.

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

[45] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271, 1999.

[46] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the strength Pareto evolutionary algorithm. In *Eurogen*, volume 3242, pages 95–100, Athens, Greece, 2001.

[47] D. W. Corne, J. D. Knowles, and M. J. Oates. The Pareto envelope-based selection algorithm for multiobjective optimization. In *International Conference on Parallel Problem Solving from Nature (PPSN VI)*, volume 1917, pages 839–848. Springer-Verlag, 2000.

[48] K. Deb, M. Mohan, and S. Mishra. Towards a quick computation of well-spread Pareto-optimal solutions. In *Second International Conference on Evolutionary Multi-Criterion Optimization*, volume 2632, pages 222–236, Faro, Portugal, 2003. Springer.

[49] A. Liefooghe, M. Basseur, L. Jourdan, and E. G. Talbi. ParadisEO-MOEO: A framework for evolutionary multi-objective optimization. In *International Conference on Evolutionary Multi-Criterion Optimization (EMO 2007)*, volume 4403, pages 386–400, Matsushima, Japan, 2007. Springer.

[50] E. Zitzler, L. Thiele, M. Laumanns, C. M. Fonseca, and V. Grunert da Fonseca. Performance assessment of multiobjective optimizers: an analysis and review. *IEEE Transactions on Evolutionary Computation*, 7(2):117–132, 2003.

[51] K. Deb. *Multi-objective optimization using evolutionary algorithms*, volume 16. John Wiley & Sons, Chichester, UK, 2001.

[52] M. Basseur, F. Seynhaeve, and E. G. Talbi. Design of multi-objective evolutionary algorithms: Application to the flow-shop scheduling problem. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2002)*, volume 2, pages 1151–1156, Piscataway, NJ, USA, 2002.

[53] H. Meunier, E. G. Talbi, and P. Reininger. A multiobjective genetic algorithm for radio network optimization. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2000)*, volume 1, pages 317–324, San Diego, USA, 2000.

[54] C. Grosan, M. Oltean, and D. Dumitrescu. Performance metrics for multiobjective optimization evolutionary algorithms. In *Proceedings of Conference on Applied and Industrial Mathematics (CAIM), Oradea*, 2003.

[55] G. S. Dulikravich, T. J. Martin, M. J. Colaço, and E. J. Inclan. Automatic switching algorithms in hybrid single-objective optimization. *FME Transactions*, 41(3):167–179, 2013.

[56] IOSO NM Version 1.0, User's Guide. IOSO Technology Center, Moscow, Russia, 2003.

[57] R. Jin, W. Chen, and T. W. Simpson. Comparative studies of metamodeling techniques under multiple modeling criteria. In *Proceedings of the 8th AIAA / USAF / NASA / ISSMO Multidisciplinary Analysis & Optimization Symposium*, Long Beach, CA, 2000.

[58] M. D. McKay, W. Conover, and R. J. Beckman. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 21(2):239–245, 1979.

[59] I. M. Sobol. On the distribution of points in a cube and the approximate evaluation of integrals. *USSR Computational Mathematics and Mathematical Physics*, 7(4):86–112, 1967.

[60] G. S. Dulikravich and M. J. Colaço. *Hybrid optimization algorithms and hybrid response surfaces*. Springer, 2015.

[61] R. H. Myers, D. C. Montgomery, and C. M. Anderson-Cook. Response surface methodology: product and process optimization using designed experiments, 2009.

[62] G. Matheron. Principles of geostatistics. *Economic Geology*, 58(8):1246–1266, 1963.

[63] R. L. Hardy. Multiquadric equations of topography and other irregular surfaces. *Journal of Geophysical Research*, 76(8):1905–1915, 1971.

[64] S. S. Haykin. *Neural networks and learning machines*, volume 3. Pearson Upper Saddle River, NJ, USA, 2009.

[65] R. J. Moral and G. S. Dulikravich. A hybridized self-organizing response surface methodology. In *12th AIAA/ISSMO Multidisciplinary Analysis and Optimisation Conference*, Victoria, BC, Canada, 2008.

[66] J. A. F. Prieto and J. R. V. Perez. Design of an adaptive genetic algorithm for maximizing and minimizing throughput in a computer network. In *23rd International Symposium on Computer and Information Sciences (ISCIS'08)*, pages 1–4. IEEE, 2008.

[67] A. A. Adewuya. New methods in genetic search with real-valued chromosomes. Master's thesis, Massachusetts Institute of Technology, Cambridge, 1996.

[68] Z. Michalewicz. *Genetic algorithms + Data structures = Evolution programs (3rd Ed.)*. Springer-Verlag, London, UK, 1996.

[69] E. Alba. Parallel evolutionary algorithms can achieve super-linear performance. *Information Processing Letters*, 82(1):7–13, 2002.

[70] V. Donaldson, F. Berman, and R. Paturi. Program speedup in a heterogeneous computing network. *Journal of Parallel and Distributed Computing*, 21(3):316–322, 1994.

[71] W. Rivera. Scalable parallel genetic algorithms. *Artificial Intelligence Review*, 16(2):153–168, 2001.

[72] V. S. Gordon and L. D. Whitley. Serial and parallel genetic algorithms as function optimizers. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 177–183, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[73] E. Cantú-Paz. Designing efficient master-slave parallel genetic algorithms. IlliGAL Report No. 97004, 1997.

[74] D. S. Knysh and V. M. Kureichik. Parallel genetic algorithms: a survey and problem state of the art. *Journal of Computer and Systems Sciences International*, 49(4):579–589, 2010.

[75] M. Ivanovic, V. Simic, B. Stojanovic, A. Kaplarevic-Malisic, and B. Marovic. Elastic grid resource provisioning with WoBinGO: A parallel framework for genetic algorithm based optimization. *Future Generation Computer Systems*, 42:44–54, 2015.

[76] E. Cantú-Paz. A survey of parallel genetic algorithms. *Calculateurs Paralleles, Reseaux et Systems Repartis*, 10(2):141–171, 1998.

[77] S. Shan and G. G. Wang. Survey of modeling and optimization strategies to solve high-dimensional design problems with computationally-expensive black-box functions. *Structural and Multidisciplinary Optimization*, 41(2):219–241, 2010.

[78] N. Marco and S. Lanteri. A two-level parallelization strategy for genetic algorithms applied to optimum shape design. *Parallel Computing*, 26(4):377–397, 2000.

[79] V. E. Bazterra, M. Cuma, M. B. Ferraro, and J. C. Facelli. A general framework to understand parallel performance in heterogeneous clusters: analysis of a new adaptive parallel genetic algorithm. *Journal of Parallel and Distributed Computing*, 65(1):48–57, 2005.

[80] E. Cantú-Paz. *Efficient and accurate parallel genetic algorithms*, volume 1. Springer Science & Business Media, 2000.

[81] L. Zheng, Y. Lu, M. Guo, S. Guo, and C. Z. Xu. Architecture-based design and optimization of genetic algorithms on multi- and many-core systems. *Future Generation Computer Systems*, 38:75–91, 2014.

[82] M. Golub and L. Budin. An asynchronous model of global parallel genetic algorithms. In *Second ICSC Symposium on Engineering of Intelligent Systems (EIS2000)*, 2000.

[83] M. Yagoubi, L. Thobois, and M. Schoenauer. Asynchronous evolutionary multiobjective algorithms with heterogeneous evaluation costs. In *2011 IEEE Congress of Evolutionary Computation (CEC)*, pages 21–28, 2011.

[84] P. Nordin, R. Braun, and P. Krus. Job-scheduling of distributed simulation-based optimization with support for multi-level parallelism. In *Proceedings of the 56th Conference on Simulation and Modelling (SIMS 56), October, 7-9, 2015, Linköping University, Sweden*, number 119, pages 187–197. Linköping University Electronic Press, 2015.

[85] M. Munetomo. Realizing robust and scalable evolutionary algorithms toward exascale era. In *2011 IEEE Congress of Evolutionary Computation (CEC)*, pages 312–317, 2011.

[86] M. S. Eldred, W. E. Hart, B. D. Schimel, and B. G. van Bloemen Waanders. Multilevel parallelism for optimization on MP computers: Theory and experiment. In *Proc. 8th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, number AIAA-2000-4818, Long Beach, CA*, volume 292, pages 294–296, 2000.

[87] K. De Jong. *An analysis of the behaviour of a class of genetic adaptive systems*. PhD thesis, University of Michigan, 1975.

[88] A. V. Levy, A. Montalvo, S. Gomez, and A. Calderon. Topics in global optimization. Lecture Notes in Mathematics Vol. 909, New York, 1982.

[89] H. P. Schwefel. *Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie*, volume 1. Birkhäuser, Basel, Switzerland, 1977.

[90] S. Luke. *Essentials of metaheuristics*. Lulu Com, 2013.

[91] Intel® Math Kernel Library (Intel® MKL). https://software.intel.com/en-us/intel-mkl.

[92] B. A. Wichmann and I. D. Hill. Algorithm AS 183: An efficient and portable pseudo-random number generator. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 31(2):188–190, 1982.

[93] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[94] M. Saito and M. Matsumoto. SIMD-oriented Fast Mersenne Twister: a 128-bit pseudorandom number generator, 2008.

[95] P. Bratley, B. L. Fox, and H. Niederreiter. Implementation and tests of low-discrepancy sequences. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 2(3):195–213, 1992.

[96] P. Hellekalek. Good random number generators are (not so) easy to find. *Mathematics and Computers in Simulation*, 46(5):485–505, 1998.

[97] Paradiseo, a software framework for metaheuristics. http://paradiseo.gforge.inria.fr.

2

Implementation of a new optimization library: Optimus

Abstract. The aim of this second chapter is to explain how the implementation of the new optimization library Optimus has been carried out, based on the research on optimization theory conducted in Chapter 1. The design requirements are defined, after which a state of the art study on currently available optimization libraries is presented. Once the development strategy of the new library is fixed, the main features of Optimus are introduced. The final validation tests prove the suitability of the new library for solving real-world optimization problems.

2.1 Introduction

2.1.1 General design requirements of a library

The aim of the work presented in this chapter is the creation of a new optimization framework, i.e. a set of classes that embody an abstract design for solutions to a family of related problems [1]. However, writing a new library is always a tough and time consuming task. Therefore, it is necessary to take a while for thinking of a good conceptual design and its appropriate implementation. This allows obtaining a well-structured and extendable code, easing future maintenance and making the use of the library more comfortable for other researchers. A good review of key design aspects may be found in [1]. The main features expected from a library are summarized hereafter:

- **Code reusability:** Reusability may be defined as the ability of software components to build many different applications. Code development is time consuming and error-prone, what makes development of new code from scratch every time a new problem arises highly undesirable. On the contrary, the use of thoroughly tested and well-documented libraries can save considerable time and headaches. It must be said that the object-oriented paradigm is particularly well-suited to develop reusable, flexible and extendable libraries by means of a hierarchical set of classes.

- **Conceptual separation between the solution method and the problem:** The solution method must be abstract enough to be able to solve similar but distinct problems.

- **Code correctness and reliability:** The developed code must be free of bugs, provide detailed error messages, avoid deadlocks at run time, have well tested algorithms, etc.

- **An adequate programming language:** It is essential to choose the language that best suits the requirements of parallel and distributed programming, other software expected to be coupled to the framework, performance, etc.

- **Portability:** The framework must be deployable on platforms with variable architectures (networks of PCs and workstations, massively parallel machines, etc.) and their associated operating systems. Therefore, it is important to code using a portable language and standard libraries.

- **Performance and efficient parallelization:** The good performance of the code is a must due to the high computational cost of simulations. Moreover, an efficient and easy-to-handle parallelization is desirable in order to take advantage of High Performance Computing (HPC) resources.

- **Ease-of-use:** The user-friendliness of the framework must guarantee access to full functionality with a minimum effort. This involves the implementation of a graphical user interface, simulation monitoring, documentation, etc.

2.1.2 Specific design requirements of the new optimization library

Apart from the already mentioned general design requirements, each new development involves additional specific characteristics. In agreement with the objectives of this Doctoral Thesis, these specific features of the new library are listed hereafter:

- The aim of the Doctoral Thesis is the development of a generic mathematical optimization tool, applicable in any field of science and engineering. Nevertheless, being this research activity hosted by the Heat and Mass Transfer Technological Center (CTTC), a special focus is to be put on the application of the library to the fields of expertise of the Center: Computational Fluid Dynamics and Heat Transfer (CFD & HT), multi-physics simulation, etc.

- CTTC is developing its own software called TermoFluids [2] since several years ago. It is a must for the new library to be compatible with the software packages belonging to TermoFluids and to provide an appropriate coupling interface. The interaction between libraries may be motivated either because the new optimization framework needs to access basic in-house libraries or because an optimization problem is defined using in-house CFD & HT libraries.

- CFD & HT is a field of engineering characterized by the need of solving huge non-linear equations systems involving a high computational cost. Consequently, the use of High Performance Computing (HPC) infrastructures is a common practice. The new optimization library must be designed for maximizing its performance in such equipment, being portability a key aspect to ensure the usability of the code in worldwide supercomputers. Moreover, the irruption of the massively parallel computing paradigm is to be taken into account in the framework's design.

- The new library must be subject to the coding standards at CTTC. This involves

using C++ as object-oriented programming language and MPI (Message Passing Interface) as the communication standard for parallel computing.

- The development of a new library is always a tough task. The use of third-party libraries is allowed in case it is considered useful in terms of shortening the project's duration or enhancing the quality of the final implementation. However, the following restrictions are to be taken into account: i) only open-source libraries protected by a GNU LGPL or a less restrictive license are accepted, i.e. CTTC must be free to independently decide the most appropriate license for the new optimization framework, ii) only libraries written in C or C++ are accepted, iii) the new library must hold the core algorithms in order to allow changing any feature of the behavior of the optimizer on demand.

2.1.3 Computing facilities at CTTC

Although portability has already been highlighted as a key feature of the new code, its development and testing have been carried out using the supercomputing facilities available at CTTC. The information of both available computer clusters is attached hereafter (see also Fig. 2.1):

- **Cluster JFF2 (dating from 2009):** This Beowulf HPC cluster called Joan Francesc Fernàndez 2nd Generation (JFF2) has 128 cluster nodes, each node counting 2 AMD Opteron Quad Core processors with 16 Gigabytes of RAM memory. The nodes are linked with an infiniband DDR 4X network interconnection with latencies of 2.6 microseconds and a 20 Gbits/s bandwidth.
- **Cluster JFF3 (dating from 2011):** This Beowulf HPC cluster JFF third generation has 40 cluster nodes, each node counting 2 AMD Opteron with 16 Cores for each CPU linked with 64 Gigabytes of RAM memory and an infiniband QDR 4X network interconnection between nodes with latencies of 1.07 microseconds and a 40 Gbits/s bandwidth.

The operating system in service is CentOS 6.5, and OpenMPI 1.8.5 is the used implementation of the MPI-3 standard for communications.

Figure 2.1: UPC's JFF supercomputer consists of 168 computer nodes with 2024 cores and 4.6 TB of RAM in total.

2.1.4 Concluding remarks

The development of the new optimization framework has to be carried out taking into account all requirements stated in the previous sections. Most of them are common features expected from a library. Regarding the specific requirements, they may be summarized by saying that the new library must perfectly fit the current optimization needs of CTTC and be fully compatible with the TermoFluids software.

For finishing this section dedicated to design concepts, some concluding remarks are listed below:

- According to the specific design requirements, the programming language to be used is C++ and the communications standard for parallel computing is MPI (Message Passing Interface). C++ is a widespread, portable and performant object-oriented language. Although Java, for instance, includes a lot of interesting concepts related to parallelism, its overall performance is worse. MPI is also a widely used and portable communication standard, which provides a convenient mechanism for modularizing parallelism through the use of "communicators"

[3]. Thus, both C++ and MPI seem to be good choices for the new optimization framework.

- Fulfilling the portability requirement means that the code should run in different computer architectures and operating systems. For the development of this Doctoral Thesis, it has been considered enough that the code runs with the hardware and software stated in the previous section.
- The coupling between the optimization framework and the models to be optimized has not been defined. As a first and most universal approach, it has been decided to use a direct coupling, i.e. the model to be optimized will behave as a black box for the optimizer, receiving values for the optimization variables from the optimizer and returning the values of the objective functions.
- The user-friendliness of the framework is an important feature. However, it is easier to first focus on the algorithmic of the library and to take care of user-friendliness once an acceptable development level of the library has been reached. Thus, the creation of a graphical user interface, documentation, etc. have not been considered in the scope of this Doctoral Thesis.
- The possibility of using third-party libraries has been considered crucial due to the limited time available for the completion of the Doctoral Thesis, the scope of the topic and the fact that optimization theory has been studied for several decades.
- The optimization framework will include single-objective and multi-objective capabilities for real-valued problems. In the scope of this Doctoral Thesis, a global search method (a genetic algorithm) and a local search method (a gradient method) will be implemented. Discrete-valued problems are considered of less importance for the kind of CFD & HT studies carried out at CTTC, so operators for discrete optimization will not be implemented for the moment.

2.2 State of the art of optimization libraries

2.2.1 Description of the project's needs

The possibility to reuse already existing code is a good chance of accelerating the development of the new optimization framework. Plenty of scientific and mathematical

software has been released in the last years, easily accessible thanks to Internet. Part of this development has been supported by governmental initiatives through public research funding. However, the considerable amount of available software makes selection a hard task unless the project's needs are well defined.

The available software may be divided in proprietary software (usually commercial software) and open-source software (usually free software). The advantages of proprietary software are that the distributor offers a guarantee and technical support to the customer, and also that the implemented algorithms are usually a result of already settled knowledge. The main disadvantage is that the customer is unable to see the source code and, of course, a certain amount of money is to be paid for the license. On the other hand, open-source software is usually distributed with no guarantee or technical support, but the code is for free and may be accessed with no restriction.

It was explained in the previous section that only open-source software with a GNU LGPL or equivalent license is accepted to be used in this project. Fortunately, many open-source initiatives for developing mathematical and scientific libraries have already been carried out and may be of great value. Proprietary software cannot be used for the development of the new optimization framework, but it can serve as a benchmark for validation tests and offers a good description of the interfaces and state of the art to which the research community and industry are used to.

The needs expected to be fulfilled by means of a third-party open-source library, and consequently the selection criteria, are the following:

- A reference library containing a genetic algorithm for both single-objective and multi-objective real-valued optimization is needed.
- A reference library containing a gradient-based single-objective local search method is needed.
- Additional features like other optimization algorithms or mathematical methods will be positively evaluated, provided that the quality of the required algorithms (a genetic algorithm and a gradient-based method) is not decreased.
- All libraries must be written in C or C++ and compile under Linux operating system.
- A good object-oriented structure, abstraction level and readability will be appreciated.

- Although no guarantee may be expected from open-source software, the developer of the selected library must be a trustworthy researcher or research institution.

2.2.2 Free open-source software

References to several interesting open-source software packages have been found in the literature. In [4] (published in 2007) a comparison of 6 multi-objective optimization frameworks is provided according to the following criteria: available metaheuristic(s), framework type (black-box or white-box), available metrics, available hybrid algorithms, programming language and parallel features. [5] (published in 1999) gives an overview of sequential and parallel genetic algorithms that existed at the time. [1] (published in 2004) presents a list of other 6 existing optimization frameworks. Although no more articles will be cited here, additional references to optimization software packages may be surely found. However, the articles mentioned before were written some years ago and may not be representative of the last advances in the field of mathematical optimization.

A search has been carried out in the Internet seeking for more up-to-date open-source software. A list with the most promising codes that were found, together with a short review, is attached hereafter.

Evolving Objects (EO) [6] / Parallel and Distributed Evolving Objects (Paradiseo) [1,7]
EO and its further evolution Paradiseo are template-based, ANSI-C++ evolutionary computation libraries which help write stochastic optimization algorithms very fast. The framework is the result of a European joint work and allows finding solutions to all kind of hard optimization problems, from continuous to combinatorial ones. Its main characteristics are a flexible object-oriented design, portability, availability of evolutionary (including genetic algorithms) and discrete local search methods and parallelization options. The framework's development was active until 2013 and took place at the University of Granada first and at INRIA (Institut National de Recherche en Informatique et en Automatique) finally. Good documentation is available and the library has a GNU LGPL like license. It can be compiled under Linux.

GAlib [8]
It is a set of C++ genetic algorithm objects developed at the Massachusetts Institute of Technology (MIT). GAlib is known to be a mature code with very efficient operators and

good documentation. The library may be compiled under Linux and seems to have a nice graphical interface. The original source code copyright is owned by MIT, but allows modification and licensing fulfilling some restrictions. The last release of the code took place in 2007.

GAUL [9]
The Genetic Algorithm Utility Library is a flexible programming library that implements genetic algorithms, apart from other stochastic, evolutionary and local search methods. The code has been designed for Linux, written in C language and parallelized using several standards, including MPI. The last release of the library dates from 2009. GAUL is a trademark of Stewart Adcock and is distributed under the GNU GPL license.

GENEVA [10]
Geneva is a software library which enables users to solve large scale optimization problems in parallel on devices ranging from multi-processor machines over clusters to Grids and Cloud installations. It currently supports evolutionary algorithms (including genetic algorithms), swarm algorithms, gradient descents and a form of simulated annealing. Performance and extensibility are at the core of Geneva's object-oriented design. The code is written in C++, may be compiled in Linux environments and is distributed under a GNU Affero GPL v3 license, although there are additional licensing options available. Good documentation is provided with the software, whose last release dates from 2015. Geneva was developed and is maintained by Gemfony scientific, a spin-off from Karlsruhe Institute of Technology (KIT).

DAKOTA [11]
The DAKOTA toolkit is a software framework for systems analysis, encompassing optimization, parameter estimation, uncertainty quantification, design of computer experiments, and sensitivity analysis. It interfaces with a variety of simulation codes from a range of engineering disciplines, and it manages the complexities of a broad suite of capabilities through the use of object-oriented abstraction, class hierarchies, and polymorphism. The library includes many algorithms (also genetic algorithms), is written in C++, implements the MPI standard and may be compiled under Linux. The code has been developed at Sandia National Laboratories and its distribution is restricted by the GNU LGPL license. The last available version was released in 2016.

TRILINOS [12]
The Trilinos Project is an effort to facilitate the design, development, integration and ongoing support of mathematical software libraries. That effort is particularly focused on developing parallel solver algorithms and libraries within an object-oriented software framework for the solution of large-scale, complex multi-physics engineering and scientific applications. Among the many packages conforming Trilinos, attention has been put on MOOCHO (Multifunctional Object-Oriented arCHitecture for Optimization). It is designed to solve large-scale, equality and inequality nonlinearly constrained, non-convex optimization problems (i.e. nonlinear programs) using reduced-space successive quadratic programming (SQP) methods. The library is written in C++ and may be compiled under Linux. The code has been developed at Sandia National Laboratories and its distribution is restricted by the GNU LGPL license. The last available version was released in 2016.

2.2.3 Proprietary commercial software

The search of proprietary software has been less intensive because it does not provide a source code which may be reused for creating the new optimization framework, and also because its use as a benchmark is not foreseen in the scope of this Doctoral Thesis. Nevertheless, a few renowned commercial codes are introduced next.

IOSO [13]
Indirect Optimization Based Upon Self-Organization (IOSO) is a new generation multi-dimensional nonlinear optimization software based on the response surface technology. Its strategy differs significantly from other well-known approaches to optimization, apparently increasing the efficiency and robustness with respect to standard algorithms. The software has been designed for dealing with heavy tasks with up to 100 variables and 20 objectives, and provides full-automatic optimization algorithms which do not need to be tuned up by the user. It also provides coupling interfaces with the most widespread CAD, CFD and FEA codes.

MATLAB [14]
This is a well-known scientific software commercialized by Mathworks which includes plenty of mathematical tools. There are two toolboxes regarding optimization, namely the Optimization Toolbox and the Global Optimization Toolbox. The Optimization Toolbox provides functions for finding parameters that minimize or maximize objectives

while satisfying constraints, including solvers for linear programming, mixed-integer linear programming, quadratic programming, nonlinear optimization, and nonlinear least squares. The Global Optimization Toolbox provides methods that search for global solutions to problems that contain multiple maxima or minima. It includes global search, multi-start, pattern search, genetic algorithm, and simulated annealing solvers.

LINGO [15]
This is a tool designed for building and solving linear, nonlinear (convex & nonconvex/-global), quadratic, quadratically constrained, second order cone, semi-definite, stochastic, and integer optimization models. LINGO provides a completely integrated package that includes a powerful language for expressing optimization models, a full featured environment for building and editing problems, and a set of fast built-in solvers. Apparently, the optimization software developed by Lindo Systems Inc. is in use at over half the Fortune 500 companies in the US, including 23 of the top 25.

2.2.4 Concluding remarks

After having reviewed the state of the art concerning optimization software, it is time for deciding which libraries will be taken as a reference for the development of the new framework. The main features of the 6 previously mentioned open-source libraries are compared in the following lines.

GAUL and EO/Paradiseo are similar software packages, but GAUL has the drawback of being protected by a GNU GPL license and of being written in C, not in C++. Its website affirms that the code is used in several universities, although no list of such institutions is provided. Moreover, the last release of the code took place in 2009, four years before the last release of EO/Paradiseo. Thus, it is understood that EO/Paradiseo outperforms GAUL's capacities and the latter is discarded.

GAlib seems to be a robust library, is written in C++ and has a graphical user interface. However, its copyright is owned by MIT, the last release was in 2007 and only includes genetic algorithms. Since a wider scope is covered by other packages, which are also subject to less restrictive licenses, GAlib is discarded.

GENEVA offers many interesting features: good documentation, several optimization methods, software parallelization, etc. The only drawback at first glance is its GNU GPL license. EO/Paradiseo holds many interesting characteristics as well: genetic algorithms, local search methods, parallelization, good documentation and tutorials, etc. It is also written in C++, seems to be well structured and is subject to the GNU LGPL

license, which is less restrictive than GENEVA's license. Since apparently there is no big qualitative difference between these two libraries, EO/Paradiseo is preferred due to lower licensing restrictions, so GENEVA is discarded.

DAKOTA is a framework composed by several software packages, among which there is one specialized in genetic optimization. However, DAKOTA's features for evolutionary optimization seem scarcer than those of EO/Paradiseo. Moreover, the extension and complex structure of the library could make its manipulation more complicated. Hence, EO/Paradiseo is preferred instead of DAKOTA, and the latter is discarded.

The last package to be analyzed is Trilinos. It contains many mathematical algorithms, but unlike previous libraries, it does not have any sub-package dedicated to global optimization. The only sub-package for local optimization is MOOCHO (Multifunctional Object-Oriented arCHitecture for Optimization), an object oriented C++ code for solving equality and inequality constrained nonlinear programs (NLPs) using large-scale gradient-based optimization methods. This framework seems to contain better gradient-methods than any other library mentioned until now and is distributed under the GNU LGPL license, so it may be a good complement for some other software selected for its global optimization capabilities.

After these considerations, it was decided to base the development of the new optimization framework on two open-source libraries: Paradiseo and Trilinos/MOOCHO. Both of them fulfill the necessary requirements, offer an interesting variety of global and local optimization algorithms, are well documented and are distributed under the GNU LGPL license. Paradiseo is expected to provide the main structure of the new optimization framework, as well as most of the code needed to implement a genetic algorithm. It may also be an interesting source of parallelization concepts, since it includes 3 distributed models: the island asynchronous cooperative model, the parallel/distributed population evaluation and the distributed evaluation of a single solution. These models are implemented by means of the standard libraries PThreads, PVM and MPI. Paradiseo also offers the possibility of hybridizing the genetic algorithm with one of the following discrete local search methods: hill-climbing, tabu search, simulated annealing, iterated local search, variable neighborhood search or random walk. However, the new optimization library will be focused on real-valued optimization and the lack of suitable local search methods is the major shortfall of Paradiseo. This shortfall will be covered by the gradient-methods in Trilinos/MOOCHO.

The main characteristics of the new optimization framework, which have been defined by combining the most interesting features of the two selected open-source codes, are described in the next section.

2.3 Main features of Optimus

2.3.1 Development strategy

The effort required for the creation of a new optimization library encompassing all known well-performing features and algorithms from literature exceeds by far the goals of a Doctoral Thesis. Even if the development is based on already existing software packages, each library has its particular shortcomings which are to be detected and fixed. Moreover, familiarization with a new open-source code is a time consuming task, strongly dependent on the quality of the available documentation and the programming style. Consequently, the new optimization framework is considered a first step that will help incorporate optimization technology into CTTC's research activity.

The development strategy has consisted on selecting the minimum amount of external libraries that enable all basic functionality of genetic algorithms and gradient-based local search methods. The new Optimus code holds the core structure of the optimization library and is linked to external software packages in order to incorporate certain enhancements. The development of in-house algorithms is foreseen in case this will significantly improve the code's performance for its application to CTTC's research lines. In the long run, in-house algorithms are expected to gradually replace external functions as CTTC gains experience in optimization.

The features available in Optimus are described in the following sections. A genetic algorithm is the main search engine, and is able to switch to a local search method every time stagnation of the best individual of the population is detected. The reader is referred to the scientific literature cited in Chapter 1 for theoretical foundations and an extended description of the implemented algorithms.

2.3.2 Definition of the optimization problem

The use of real-valued individuals has been implemented, being discrete (i.e. combinatorial) optimization not possible at present. Each individual may have a unique real-valued objective (single-objective optimization) or a vector of real-valued objectives (multi-objective optimization). It is possible to maximize or minimize each objective.

An interesting feature taken from Paradiseo, mentioned here because it affects the structure of individuals, is the use of self-adaptive mutation parameters. Mutation is the genetic operator that acts on a single individual altering a certain amount of genes. In particular, normal mutation modifies a gene by substituting it by a random number

according to a Gaussian distribution centered in the original value and shaped by a standard deviation parameter. At the beginning of the optimization, it is advisable to allow high standard deviation values. However, low values are more suitable as the population evolves, enabling the exploration of small regions. This behavior is known as self-adaptive mutation and is accomplished by storing the evolving standard deviation values in each individual.

Three types of self-adaptive mutation have been implemented:

- **Isotropic mutation:** A single standard deviation parameter is stored in each individual and is applied every time a normal mutation takes place on any optimization variable of that individual.
- **Anisotropic mutation [16]:** A standard deviation parameter is stored for each optimization variable, evolving each parameter independently from all others.
- **Correlated mutation [17, 18]:** In addition to as many standard deviation parameters as for the anisotropic mutation, this strategy assigns a full correlation matrix to each individual. This allows taking into account the interaction between genes regarding the objective function, and the values of the standard deviation parameters become dependent on each other.

The penalty method has been implemented as unique constraint handling strategy. The fitness of the designs that violate constraints is artificially worsened in this technique, with the aim of forcing the optimization algorithm to abandon that search region of the solutions space. This was the preferred algorithm due to ease of implementation, although the existence of better performing methods is known. Their inclusion in Optimus will be considered in the future.

A key feature of the optimization framework is the possibility of coupling it to an external objective functions evaluator. This capability is of major importance because it allows fitness evaluation by means of any third-party software, provided that it can be interfaced using C++. In the particular case of CTTC, such a coupling is necessary since the models subject to optimization will be usually built using TermoFluids, the in-house CFD & HT software coded in C++.

Two kinds of coupling interfaces must be distinguished: generic interfaces and specific interfaces. The generic interface simply gives access to certain variables of Optimus to the end user, who is responsible of getting the data contained in those variables and transmitting them to the function evaluator. The variables are the following:

- **Optimization variables:** It is a vector of double type variables containing values of the optimization variables (genes) of each individual.
- **MPI communicator:** In case the user selects to run a parallel evaluation of each individual, Optimus assigns a group of processors to each of them, creates an MPI communicator and makes it available to the objective function evaluator.
- **Directory:** It is a string variable containing the name of the directory in which files created during the evaluation of an individual (if any) may be saved. Having such directory names is important because it is not strange that individuals being evaluated simultaneously create files with identical names, thus causing file overwriting in case these are not carefully separated in different directories for each individual.

Specific interfaces transmit the same information as generic interfaces, but may be provided in order to ease couplings to the user or to adapt to some interface requirement of the software in charge of evaluating the objective function. No specific coupling interface has been developed excepting for NEST [19], a multi-physics library developed at CTTC. More specific interfaces will be added in the future if needed.

2.3.3 Single-objective vs. Multi-objective optimization

The genetic algorithm used for solving single-objective and multi-objective optimizations is similar regarding most characteristics: operators, stopping criteria, parallelization, etc. The main difference lies on how the concepts of fitness assignment, diversity preservation and elitism are handled.

Fitness assignment is a trivial task when individuals are being optimized according to a single objective. However, it was shown in Chapter 1 that several methods have been proposed for multi-objective optimization problems. If a scalar approach is desired (reducing the multi-objective problem to a single-objective problem), the single-objective optimization interface may be used. The alternative is to choose a dominance-based approach to classify multi-objective solutions. Pareto's criterion is the only dominance criterion available in Optimus at present and has been used to implement 3 dominance-based approaches: i) dominance-rank technique, ii) dominance-count technique, and iii) dominance-depth strategy.

The concept of diversity preservation only exists in multi-objective optimization. 2 diversity assignment techniques have been implemented in Optimus, namely i) sharing,

that consists on estimating the distribution density of a solution using a so-called sharing function that is related to the sum of distances to its neighborhood solutions, and ii) crowding, which consists on estimating the density of solutions surrounding a particular point of the objective space.

A different approach is to use a quality indicator which takes into account both fitness and diversity, denoting the overall goal of the optimization process. This option is available in Optimus and two common binary quality indicators have been implemented, namely the additive ε-indicator and the I_{HD}-indicator (which is based on the hypervolume metric).

Elitism is present in both single-objective and multi-objective optimization. In the single-objective case, it consists on preserving the best known individuals between successive generations. In the multi-objective case, elitism consists on maintaining an external set (archive) that allows storing either all or a subset of non-dominated solutions found during the search process. An archive class (either of bounded or unbounded size) able to store every non-dominated point of the simulation has been implemented in Optimus. It is up to the user the decision of how often the archive is to be updated and stored in the hard drive.

Nevertheless, an important difference between Paradiseo and Optimus is that the user is not expected to define all these options independently in the latter framework. 3 preconfigured options packages have been included in the code, representing the 3 most common multi-objective optimization algorithms used nowadays: NSGA-II, SPEA2 and IBEA. The user just needs to choose one of them so that all options are automatically set (see Table 2.1). In case of using IBEA, the binary quality indicator is to be explicitly chosen by the user.

Options / MO algorithms	**NSGA-II**	**SPEA2**	**IBEA**
Fitness assignment	Dominance-depth	Dominance-count and rank	Binary quality indicator
Diversity assignment	Crowding distance	Nearest neighbor	None
Selection	Deterministic binary tournament	Deterministic binary tournament	Deterministic binary tournament
Replacement	Elitist replacement	Generational replacement	Elitist replacement
Archiving	None	Fixed-size archive	None
Stopping criteria	Max. number of generations	Max. number of generations	Max. number of generations

Table 2.1: Most common multi-objective optimization algorithms [20].

Common archiving and stopping criteria for each multi-objective algorithm are shown in Table 2.1, but Optimus allows them to be specified independently of the selected algorithm, i.e. archiving may be activated for every algorithm and various stopping criteria are available.

The use of metrics in order to compare different sets of solutions is another characteristic of multi-objective optimization. Optimus is able to calculate contribution and entropy metrics on run time. Additional metrics, if needed, are to be calculated using the output files generated once the execution of Optimus has finished. The analysis may be done by means of the GUIMOO software [4], for instance.

A noticeable detail in Paradiseo is that the single-objective and multi-objective codes were developed separately, although they follow some common standards. This assertion is based on the fact that not every feature implemented for single-objective algorithmic, although conceptually valid no matter the number of objectives, is available for multi-objective algorithmic and vice-versa. One of those features is the self-adaptive mutation, already described in the previous section. In Optimus, the self-adaptive mutation has been made available for both single-objective and multi-objective algorithms.

2.3.4 Genetic operators

The evolution engine and thus the quality of the obtained optimal solution are conditioned by the selection and parametrization of genetic operators. The concept of self-adaptive mutation has already been introduced, which is accomplished by storing the evolving standard deviation values in the individual's structure. Let us call individuals carrying standard deviation parameters self-adaptive individuals, whereas individuals containing only optimization variables will be called standard individuals. The use of self-adaptive mutation provides a superior performance to the optimizer, so both the single-objective and multi-objective algorithms implemented in Optimus support self-adaptive individuals. In fact, the user is expected to use the self-adaptive method as default configuration of the optimizer. The use of standard individuals is not foreseen for solving real-world optimization problems and is only supported by the single-objective algorithm, mainly due to academic purposes.

The crossover, mutation, selection and replacement operators implemented in Optimus are described next.

Crossover

Crossover consists on recombining the genetic material of n parent individuals, although no more than 2 parents are used in Optimus. Several crossover operators have been proposed in the literature, and it is not usually known which one performs best for a certain optimization problem. This is why the possibility of combining various methods is offered in Optimus, with the aim of using the strengths of each method. The user just needs to define the desired crossover operators and to assign them a probability to be selected. Every time the genetic algorithm needs to cross over the chromosomes of two parents, one among the available crossover operators will be called by means of a roulette wheel selection.

3 crossover operators are implemented in Optimus for standard real-valued individuals:

- ***Hypercube crossover:*** Offspring are uniformly generated on the hypercube whose diagonal is the segment joining both parents, i.e. by doing linear combinations of each variable independently. The user provides an alpha parameter at the beginning of the optimization, and the crossover operator generates uniformly a random number in the range [alpha,1+alpha] for each variable. This random number is the coefficient used for defining the linear combination of the parents' values of that variable.

- ***Segment crossover:*** Offspring are uniformly generated on the segment joining both parents, i.e. the operator constructs two linear combinations of the parents with a single random number uniformly generated in the range [alpha,1+alpha]. Alpha is a parameter provided by the user at the beginning of the optimization, and the random number is the coefficient used for defining the linear combination of the parents' values for all variables.

- ***Uniform crossover:*** This operator simply exchanges values of variables between the 2 parents, creating new offspring.

In the case of using self-adaptive individuals, two crossover operators are to be selected for recombining 2 individuals: the first operator defines how to recombine the optimization variables of the individuals, whereas the second operator defines how to recombine the self-adaption parameters. The crossover operators available are uniform crossover and hypercube crossover (with the alpha coefficient fixed to 0 value), being the use of segment crossover not enabled for self-adaptive individuals. The roulette-wheel

selection between various crossover operators is neither enabled, so a single operator is to be chosen for each part of the individual.

Two parents are needed to apply a standard crossover operator in Optimus, either when standard or self-adaptive individuals are used. However, an additional option called global crossover is available for self-adaptive individuals, which consists on randomly selecting two parents for each gene of the offspring that will be created.

Mutation

Mutation consists on altering a certain percentage of genes of each offspring created by the crossover operator. Since the best performing operator is not usually known (as it happens in the case of crossover operators), Optimus offers the possibility to combine various methods to use the strengths of each. The user activates the desired mutation operators and assigns to each of them a probability to be selected. Every time the genetic algorithm needs to mutate the chromosome of an offspring, one among the available operators is chosen by means of a roulette wheel selection.

A single mutation operator is available in Optimus for self-adaptive individuals:

- ***Self-adaptive mutation:*** A normal mutation is applied to each optimization variable. Each of these mutations is defined according to a normal distribution centered in the original variable's value and whose standard deviation is taken from the corresponding self-adaptation parameter carried by the individual. A mutation is also applied to each self-adaptation parameter.

3 mutation operators are implemented in Optimus for standard real-valued individuals:

- ***Uniform mutation:*** This operator modifies all variables by choosing new values uniformly on an interval centered on the old value and of width $2 \cdot epsilon$, being *epsilon* defined by the user for each optimization variable.
- ***Deterministic-uniform mutation:*** Exactly k variables are modified uniformly by choosing a new value from an interval centered on the old value and of width $2 \cdot epsilon$, being *epsilon* defined by the user for each optimization variable.
- ***Normal mutation:*** Also called Gaussian mutation, this operator acts on every optimization variable of an individual creating new values according to a normal distribution centered in the original value and with a fix standard deviation parameter defined by the user.

Selection

The selection step consists on choosing the individuals that will be used to generate the offspring population, being its size fixed at the beginning of the optimization. As it was already said, the general behavior is that the better (fitter) an individual, the higher its chance of being selected. The selection operators implemented in Optimus for single-objective optimization (either with standard or self-adaptive individuals) are the following:

- ***Deterministic tournament:*** This operator returns the best of T uniformly chosen individuals in the population. The number of tournaments to be carried out is equal to the number of needed parents.
- ***Stochastic tournament:*** The operator chooses uniformly two individuals from the population and returns the best one with probability R (tournament rate), being the real parameter R in the range $[0.5, 1.0]$. The number of tournaments to be carried out is equal to the number of needed parents. Note that a stochastic tournament with rate 1.0 is strictly identical to a deterministic tournament of size 2.
- ***Roulette wheel:*** This is the classical selection method used by Goldberg [21] in which each parent is selected according to a probability proportional to its fitness.
- ***Ranking:*** This method starts by assigning a worth, i.e. a modified fitness value, to each individual of the population. Then selection is carried out by means of a roulette wheel algorithm based on the previously calculated worth values. Two parameters are needed for the worth calculation: the pressure (ranging in $(0, 1]$) and the exponent (always greater than 0). Worth values are contained in $[m, M]$, where $m = 2 - pressure/populationSize$ and $M = pressure/populationSize$. Inside these bounds, the spacing between worth values depends on the exponent.
- ***Ordered sequential selection:*** This operator sorts the population from best to worst and returns as many individuals as required following the list. If the population is exhausted and more individuals are needed, it loops back to the beginning of the list and continues returning individuals until the required number of parents is satisfied. If the number of required parents is smaller than the size of the source population, the best individuals are selected once. If the number required parents is N times that of the source size, all individuals are selected exactly N times.

- ***Unordered sequential selection:*** This operator shuffles the population and returns as many individuals as required. If the population is exhausted and more individuals are needed, it loops back to the beginning of the list and continues returning individuals until the required number of parents is satisfied.

The three multi-objective optimization algorithms included in Optimus (NSGA-II, SPEA2 and IBEA) use a binary deterministic tournament selection, so other selection methods have not been made available for multi-objective optimization to date.

Replacement

The replacement operator is applied after the birth of all offspring and consists on selecting the survivors from the current and offspring populations in some arbitrary way. In Optimus the population size is always kept constant from one generation to the next one, being the possibility of having a variable population size not implemented. The available replacement operators are valid for either standard or self-adaptive individuals and may be classified according to the following schemes: merge-reduce operators and reduce-merge operators. The merge-reduce scheme has two major steps, first merging both populations of parents and offspring and then reducing that big population to the right size. In the reduce-merge scheme parents are first reduced of the exact number of offspring and then merged with the offspring population. In the latter case, it is implicitly assumed that few offspring have been generated, although this is not mandatory.

3 merge-reduce operators have been implemented:

- ***Comma replacement:*** This operator, which is common in Evolution Strategies, selects the best offspring and discards all parents. Hence, at least as many offspring as the size of the population must be created.

- ***Plus replacement:*** It first merges the offspring and the parents and finally the best individuals among them become the next generation. This operator is also common in Evolution Strategies.

- ***EP tournament:*** This is a classical operator in Evolutionary Programming (EP). First the offspring and parents are merged and then a global tournament of size T begins. It works by assigning a score to all individuals in the population. Starting with a score of 0, each individual I is opposed T times to a uniformly chosen individual. I's score is incremented by 1 every time it wins, and by 0.5 every

time it draws. Once all tournaments are finished, the individuals for the next generation are selected deterministically based on their scores.

3 reduce-merge operators have been implemented:

- ***Worst replacement:*** The worst parents are replaced by all offspring.
- ***Deterministic tournament:*** Each parent to be replaced is chosen by an inverse deterministic tournament, i.e. the operator returns the worst of T uniformly chosen individuals in the population.
- ***Stochastic tournament:*** Each parent to be replaced is selected by an inverse binary stochastic tournament, i.e. each time the operator chooses uniformly two individuals from the population and returns the worst one with probability R (tournament rate), being the real parameter R in the range $[0.5, 1.0]$.

The possibility of activating weak elitism is also implemented for single-objective optimization, which means that if the best fitness in the new population is worse than the best fitness of the parent population, the worst individual of the new population is replaced by the best parent. This strategy ensures that the overall best fitness in the population will never decrease.

2.3.5 Hybrid methods

One single hybrid algorithm has been implemented in Optimus, only available for single-objective optimizations. It combines two constitutive algorithms sequentially: the genetic algorithm as global search method and the Trilinos/Moocho package as local search method.

Two approaches to sequential hybrid algorithms have been found in the literature [22]: i) the best solution found by the GA is taken as the starting point for the local search method, and ii) the gradient method can be incorporated in the GA as a new operator and can be applied either to the best individual or to the individuals corresponding to local optima. The first approach has been implemented in Optimus.

The control algorithm in charge of switching automatically from one method to the other is represented in the flow diagram in Fig. 2.2. The user must define the stagnation criterion for the GA, i.e. the maximum allowed number of generations (let us refer to it as N) to get some improvement of the best known solution so far. If no improvement is obtained after N generations, the control algorithm checks if the best individual so

far has been previously used for starting a local search. If so, it means that the local search was unable to improve it and, since it is a deterministic method, exactly the same result would be reached again. Thus, the local search is skipped and the flow returns to the genetic algorithm. If no local search was started in the past with that individual, it is launched (i.e. the Trilinos/Moocho package is called) and the obtained result is compared with the original individual. If the local search result is better, then the best individual in the population is replaced by the newly found solution. After this step, the flow returns to the genetic algorithm and continuation criteria area checked, as usual.

The Trilinos/Moocho package (Multifunctional Object-Oriented arCHitecture for Optimization) has been designed to solve large-scale optimization problems using reduced-space successive quadratic programming (SQP) methods, as it was already explained in a previous section. Moocho transforms the original problem into an equivalent quadratic problem, having both of them the same solution. The advantage obtained with the transformation is a higher convergence rate. The new problem could be solved by means of a BFGS algorithm. Nevertheless, the lBFGS algorithm is preferred instead in order to decrease the memory requirement. This issue is of special concern in the case of problems having many variables. The only drawback of lBFGS with respect to BFGS is its linear convergence rate, whereas higher convergence rates are achieved by BFGS.

No hybrid method is implemented for multi-objective optimization to date, but an expansion of the framework in this direction will be considered in the future.

2.3.6 Continuation criteria

The continuation criteria (also called stopping criteria) are in charge of controlling if the optimization process shall end or go on. Several criteria were implemented in Optimus, being possible to enable more than one in the same simulation. In that case, a single stop signal sent by any active criterion is enough to make the optimization finish.

The continuation criteria available in Optimus are described hereafter:

- Stop optimization if ...
  - Resource limitation criteria
    - ... maximum number of generations reached: When the genetic algorithm has carried out a certain number of generations, the best solution so far is returned as the global optimum.

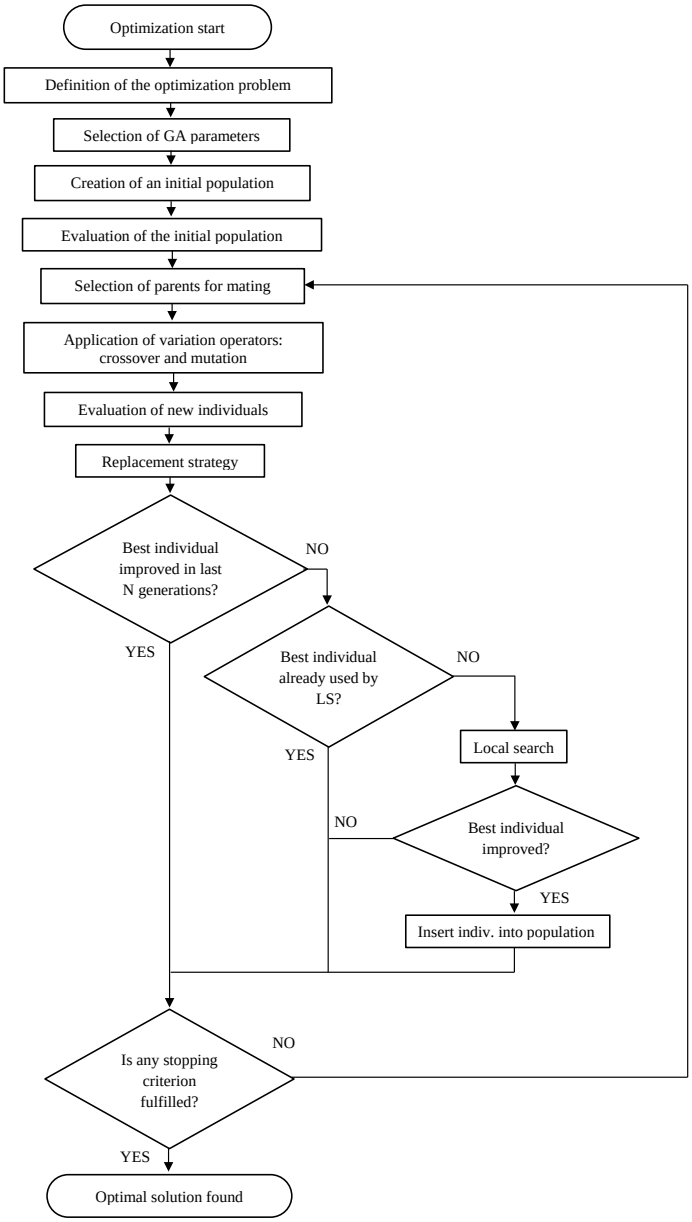


Figure 2.2: Flow chart of a sequential hybrid algorithm, composed by a genetic algorithm and a local search method.

        * ... maximum number of evaluations reached: When the optimizer has carried out a certain number of objective function evaluations, the best solution so far is returned as the global optimum.
        * ... maximum simulation time reached: When the optimization process has spent a certain amount of wall clock time, the best solution so far is returned as the global optimum.
    - Solution's stagnation criteria
        * ... best individual did not improve in N generations: When the optimizer has not been able to improve the best solution so far after a certain number of generations, that solution is returned as the global optimum.
    - Target accomplishment criteria
        * ... target fitness value reached: When the optimizer has obtained a solution with a fitness value which is equal to or better than a target fitness value established at the beginning of the optimization, the best solution so far is returned.

- Continue optimization until ...
    - Minimum usage of resources criteria
        * ... minimum number of generations reached: The optimization process cannot be stopped until the genetic algorithm has carried out a certain amount of generations, even though some stopping criterion is fulfilled.

2.3.7 Statistics

It is important to provide some information of the optimization process in run time in order to allow the engineer to check regularly the correctness of the simulation. Due to the high number of objective function evaluations usually needed, an early detection of unexpected behaviors might save a considerable amount of computational time. Moreover, the provided information must be enough to evaluate if the obtained result is the best attainable solution for the simulation that has been run.

Thus, it was decided to calculate the following data after the genetic algorithm completes each generation:

- **Number of generations:** Total amount of generations carried out so far.

- **Number of evaluations:** Total amount of objective function evaluations carried out so far.
- **Wall clock time:** Total amount of wall clock time spent so far by the optimization process.
- **Best fitness value:** Fitness value of the best individual available in the current population. In the case of multi-objective optimization, the best value available in the current population for each objective is identified.
- **Statistics of the population:** Average and standard deviation of the fitness, taking into account every individual in the current population. In the case of multi-objective optimization, the average and standard deviation of the current population for each objective is calculated.

Some additional metrics are available after the completion of each generation when multi-objective optimizations are carried out. These metrics are contribution and entropy, and are computed for both the archive and the current population.

2.3.8 Parallelization

Task management strategy

After the thorough state of the art study of parallelization techniques carried out in Chapter 1, it was decided to first implement a two-level model (see Fig. 2.3 (a)): a master-worker strategy for the optimization algorithm in the upper level, and a case dependent fine-grain parallelization of individuals in the lower level. The parallel task manager was implemented from scratch, although some interesting notes about the parallel algorithms used by Paradiseo were found in [1].

The adopted parallelization strategy is used by both the genetic algorithm and the local search method and relies exclusively on hardware parallelization, thus obtaining in a shorter time identical results as sequential algorithms. This may be very beneficial when applied to optimizations with a high computational cost, as those in the field of CFD & HT. A similar approach is described in [22], where the problem of getting an optimum shape design of aerodynamic configurations is studied. Once that satisfactory results have been achieved, a 3-level parallelization model composed by an upper coarse-grain level (island model) and an intermediate master-worker level will be implemented for the genetic algorithm, keeping the fine-grain parallelization of individuals in the lower level (see Fig. 2.3 (b)).

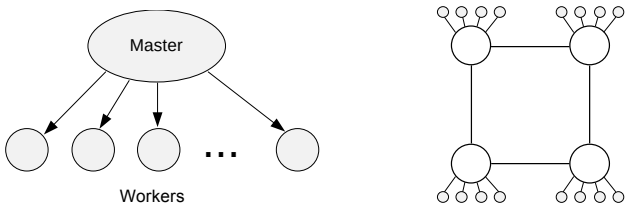


Figure 2.3: PGA models in Optimus (extracted from [5]): (a) global parallelization *(current implementation)*, (b) coarse grain + global hybrid *(future implementation)*.

The implemented 2-level strategy is similar to the self-scheduling strategy described in [23], where the master processor manages a single processing queue and maintains a prescribed number of jobs active on each group of workers. Once a group of workers has completed a job and returned its results, the master assigns the next job to this group. Thus, the workers themselves determine the schedule through their job completion speed. Heterogeneous processor speeds and/or job lengths are naturally handled, provided there are sufficient instances scheduled to balance the variation. Individuals belonging to the genetic algorithm's population compose the batch of individuals to be evaluated in parallel most of the times. Note that a synchronous genetic algorithm has been implemented, i.e. all individuals belonging to a generation are evaluated before the next generation starts. However, when the local search method is being run, a batch of individuals is evaluated every time a gradient needs to be computed.

3 parameters are expected from the user of the optimization library in order to configure the parallel task manager: the number of available processors, the number of processors that will be used to evaluate each individual, and the processor partitioning model [23]: "dedicated master" or "peer partition" approach. The master processor is dedicated exclusively to task scheduling operations in the first approach, whereas it also participates in the computation of individuals in the latter approach. Note that the peer partition approach shall be used cautiously in order to avoid inter-processor communication delays, as it is explained further on in this section.

Communications strategy

Hardware may use two types of memory: shared memory and distributed memory. On a shared memory multiprocessor, information (e.g. the population) is stored in

a common memory space and each processor is able to read and write a fraction of that information without causing any conflict. On a distributed memory computer, each processor has its own memory space and information is stored in one processor's memory. This master processor is the responsible of sending that information to the other processors (the workers) and of receiving their updates. The difference compared to a shared memory configuration is that the master has to send and receive messages explicitly. The peer partition approach is suitable for shared memory configurations, but the absence of a master dedicated exclusively to communications is prone to cause important delays when distributed memory configurations are used. The possibility of enabling communication threads has not been considered in the scope of this Doctoral Thesis.

Being MPI the selected communications standard, all data exchanges have been implemented in the form of 2-sided communications. The only exception has been the task completion control algorithm, for which 1-sided communications (also called Remote Memory Access) were preferred. The master processor creates the task completion control vector, which contains a list of every task of the batch with its status: assigned or pending. This vector is made visible to other processors by means of a memory window. If the simulation is run in a shared memory environment, the peer partition approach is preferred and thanks to the 1-sided communications style every processor is able to access the task completion control vector, blocking the access to any other processor until read/write operations are finished. However, the master shall not be locked in computation when a processor tries to access the task completion control vector in case the simulation is run in a distributed memory environment. Otherwise, the worker processor will have to wait until the master finds a while to send or receive the required information. This behavior may potentially cause important delays in message passing and it is therefore discouraged to use the peer partition approach in distributed memory machines. The only drawback of the dedicated master approach is the loss of one processor for computation, being the 1-sided communications style valid.

Task management algorithm

A task manager has been created following the previously explained 2-level master-worker parallelization strategy and taking into account all issues related to communications. Nevertheless, there is one last topic to be mentioned before presenting the task management algorithm. The execution of the genetic algorithm is conditioned by the seed provided to the random number generator, and the task scheduler also utilizes randomness for assigning individuals to groups of processors. If the same generator

instance is used for both purposes, results of sequential and parallel executions of the optimizer will differ. Hence, an additional instance of the random number generator has been included to be used exclusively by the task manager.

The implemented self-scheduling task manager is described hereafter. The main algorithm is referred to as Algorithm A, and at certain points sub-algorithms A.1, A.2, A.3 and A.4 are called.

Algorithm A *Self-scheduling task manager*

---

1. The master processor sends a message to all other processors communicating that the evaluation of a new batch of individuals (tasks) is about to start.

2. Depending on the selected configuration, the master processor may be involved in the evaluation of individuals (peer partition strategy) or will just manage MPI communications (dedicated master strategy). Processors involved in the evaluation of individuals will be referred to as "computing processors" and if the master processor exclusively manages communications, it will be called "server processor".

3. The number of tasks that may be simulated simultaneously is calculated based on the total number of computing processors and the number of processors required for simulating each task (defined by the user). After that, as many processor groups as simultaneous tasks are created, all computing processors are assigned to those groups and one processor of each group is designated as the root of the group.

4. Each group of processors creates a directory in the hard drive in order to store the files needed during the evaluation.

5. MPI communicators and memory windows are created:

   (a) A communicator is created in each group of processors for message passing between the root of the group and the rest of cores, and also for the parallel evaluation of objective functions (let us call it *groupComm*).

   (b) A communicator and some memory windows are created for connecting all group roots and the master (let us call it *rootComm*). Communications between the master and the roots may be 1-sided or 2-sided.

6. **IF** (processor is the master)
   (a) Information of all individuals of the batch is sent to every root processor.
   (b) Active and inactive cores are set (see *Algorithm A.1*).
   (c) The task evaluation loop is started (see *Algorithm A.2*).
   (d) The results of every task (objective values and simulation time) are received from root processors.
7. **ELSE IF** (processor is the root of a group of cores)
   (a) Information of all individuals of the batch is received from the master.
   (b) Active and inactive cores are set (see *Algorithm A.1*).
   (c) The task evaluation loop is started (see *Algorithm A.2*).
   (d) The results of every task (objective values and simulation time) evaluated by this group of processors are sent to the master.
8. **ELSE**
   (a) Active and inactive cores are set (see *Algorithm A.1*).
   (b) The task evaluation loop is started (see *Algorithm A.2*).
9. All MPI communicators and memory windows are freed.

Algorithm A.1 *Set active/inactive processors*

1. All processors are set to be active by default at the beginning of the evaluation of a new batch of individuals.
2. If a group of processors is composed by fewer cores than the number specified by the user, all cores forming that group are set to be inactive and will not evaluate any individual. This may happen if the total number of computing cores is not divisible by the number of cores assigned to each group. In this case, the last group holds fewer processors than the rest of the groups.

3. If there are fewer individuals in the batch than available groups of computing processors, all cores in excess groups are set to be inactive and will not evaluate any individual. This is likely to happen when the local search method is used, because the number of individuals in a batch of the genetic algorithm and in that of a local search may differ considerably.

Algorithm A.2 *Task evaluation loop*

1. **IF** (active computing processor)
   (a) **WHILE** (individuals pending to be evaluated in the batch)
      i. If it is a root processor, check the list of pending tasks (see *Algorithm A.3*).
      ii. The root processor of the group communicates to other members if the group is going to evaluate an individual.
      iii. If the group is going to evaluate an individual, all processors of the group call *Algorithm A.4*.

Algorithm A.3 *Check the task status list*

1. Get the task status list from the master processor by means of Remote Memory Access (RMA). This list contains the status of each task: assigned or pending.
2. If there are pending tasks, one of them is randomly selected to be evaluated by the group of processors.
3. Then it is communicated to the master that the task has been assigned, in order to avoid that another group of processors evaluates the same individual.

Algorithm A.4 *Run task*

1. The group of processors runs the assigned task (evaluates the assigned individual), storing necessary files in the group's directory and using the group's internal communicator (*groupComm*) for evaluating the objective function(s) in parallel.

2. The obtained objective values and the evaluation time are stored in the memory of the root processor.

2.3.9 User interface

The objective of the proposed interface, although its development is not finished yet, is to offer to the user a comfortable experience by easing the definition of the optimization problem and by enabling full control over all relevant parameters that configure the optimization algorithm. A first basic implementation of the interface has been carried out, but the enhancement to a Graphical User Interface (GUI) is planned for the future.

The design of the two user interface categories, namely the input interface and the output interface, is introduced next.

Data input interface

Two kinds of input data may be distinguished: those required for defining the optimization problem and those which represent values for the parameters governing the optimization algorithm. All these data are defined by the user utilizing the following three files:

- ***optObjEvaluator.h***
  The optObjEvaluator class is contained in this file and is expected to be redefined by the user in order to create the optimization problem. The information to be provided is the following: the number of objectives and their traits (maximize or minimize), the objective function(s), the definition of the equality and inequality constraints and their Jacobians (if necessary), and the definition of the penalty method for constraints (if necessary).
- ***General algorithm's parameter sheet***
  This file contains every parameter controlling the general behavior of the optimization algorithm grouped as follows:
  - ***General options:*** Some help options which may be activated by the user are available in this group. The seed of the genetic algorithm's random number generator is also defined here.
  - ***Type of individuals:*** In the case of single-objective optimization, it is possible to use either standard or self-adaptive individuals (for isotropic,

anisotropic or correlated mutations). In the case of multi-objective optimization, however, this group of options does not exist because only self-adaptive individuals for correlated mutations are available to date.

- ***Evolution engine:*** In the case of single-objective optimization, it is defined by fixing the size of the population, the selection and replacement methods, the number of offspring to be generated and by activating the weak elitism option if desired. In the case of multi-objective optimization, the evolution engine is set by selecting one of the predefined evolutionary algorithms (NSGA-II, SPEA2 or IBEA) and by deciding whether to use or not an archive which is updated after each generation.
- ***Genotype initialization:*** The parameters to be defined in this group are the number of genes of an individual (the number of optimization variables of the problem), the initialization bounds for the genes and, in the case of using self-adaptive individuals, the initial value for the standard deviation corresponding to each gene.
- ***Local search:*** The options available in this group are the stagnation criterion for the genetic algorithm, the numerical differentiation method of the local search (first order, second order, etc.), the step size for gradient calculation and the maximum allowed violation of the optimization variables' bounds during the gradient calculations.
- ***Output on screen:*** The user may customize the run time output information showed on screen by tuning the parameters in this group. The available data include the number of objective function evaluations carried out so far, the total simulation time, and statistics of the best individual and the whole population. In the case of multi-objective optimization, there is additional information available including the storage and visualization of an archive file, and the entropy and contributions metrics of both the archive and the population. It is possible to select the frequency with which this information is displayed on screen.
- ***Output in hard drive:*** The run time output information shown on screen may also be saved in the hard drive. By tuning the parameters in this group, the user may select the directory in which the output files are to be stored and the frequency with which run time information is to be written. The possibility of storing backups of the population from time to time is also available.

- ***Parallelization:*** The parameters to be defined are the processor partitioning model (dedicated master or peer partition strategy) and the number of processors for solving each individual.
- ***Recovery information:*** Since optimization may be a costly process, it is crucial to regularly store some recovery information from which the optimization may be restarted in case the simulation is interrupted unexpectedly. This recovery information consists of two files: a file containing all the parameters defining the optimization algorithm and another file containing the simulation results obtained so far. If the optimization is interrupted, it is possible to restart it using these two files and exactly the same result as in the uninterrupted simulation will be obtained. The user may choose the names of the two files, and also the frequency with which the file containing simulation results is saved. When a simulation is wanted to be restarted using the information of the two recovery files, the names of the files to be used are specified inside this parameter group, as well.
- ***Continuation criteria:*** The user may activate the desired continuation criteria among the available ones, which were enumerated in a previous section.
- ***Variation operators:*** The parameters contained in this group are the bounds for the optimization variables or genes, the types and application probability of crossover operators, and finally the types and application probability of mutation operators.

- ***Local search's parameter sheet***
  This file contains every parameter controlling the behavior of the local search algorithm. The Trilinos/Moocho package is the only available local search algorithm to date, so the format of its original parameter file, called *Moocho.opt* by default, is used. The parameters the user may define are the following: maximum number of iterations, maximum run time, convergence tolerance, level of detail of the generated outputs and some mathematical options specifying the behavior of the local search algorithm. Additional information is available in the Trilinos Project's documentation [12].

Data output interface

Simulation information may be extracted either onto the screen or into the hard drive. The purpose of this design is to allow the user to follow the execution of the optimization

algorithm by reading the information shown on screen and, once the simulation has finished, to have extended information saved in the hard drive for post processing and later use. The user may customize the output information, as it was explained in section *Data input interface*.

The hierarchical data structure of the output files is the following:

- Optimization case directory
  - OptimusFiles directory
  - Results directory

The user is expected to create a directory (the optimization case directory) in which every file related to the optimization case is stored. All output files created by the Trilinos/Moocho package are stored here, together with the parameter file used for recovery purposes. The subdirectory *OptimusFiles* contains as many subdirectories as individuals that can be simulated simultaneously. The files generated for solving the objective function(s) of each individual are saved in those subdirectories. The subdirectory *Results*, although it may be renamed by the user, contains the files with general simulation results and also the recovery files.

2.3.10 Other features

Some additional aspects of the Optimus library are mentioned in the following paragraphs.

Regarding the random number generator, the widely used Mersenne Twister MT19937 [24] has been implemented, already mentioned in Chapter 1 and available in the Paradiseo package.

The use of surrogate models for reducing the evaluation time of computationally expensive objective functions is very interesting. It has not been implemented due to lack of time, but it is planned in a future extension of the library.

Finally, some popular mathematical functions used as benchmarks for optimizers have been implemented. On one hand, the Rosenbrock, Rastrigin, Schwefel and Griewank functions are available for testing the single-objective optimization algorithms. On the other hand, Kursawe's function (MOP4) and Zitzler-Deb-Thiele's function N. 6 (ZDT6) were included for testing multi-objective optimization algorithms. These functions are intended to be used every time modifications are inserted into the library in order to check the correctness of the new code.

2.3.11 Optimus vs. Paradiseo

The Optimus library is clearly based on the structure and algorithms available in Paradiseo. Therefore, which is the advantage of using Optimus? The summary of the main differences between both libraries to date is the following:

- The local search methods available in Paradiseo are suitable for discrete optimization, but no alternative is provided for real-valued optimization. A gradient-based local search method was added in Optimus, making possible to build a hybrid method (genetic algorithm + gradient method) for real valued optimization problems.
- The existence of at least two development branches is very noticeable in Paradiseo. An important consequence is that the use of self-adaptive individuals is possible in single-objective optimization, but not in multi-objective optimization. Since the use of this kind of more sophisticated individuals clearly improves the obtained results, self-adaptive individuals for multi-objective optimizations have been implemented in Optimus.
- The parallelization strategy in Optimus and Paradiseo differs considerably. Paradiseo is designed focusing on combinatorics problems whose objective function evaluation times are usually low. The master-worker parallelization algorithm in Optimus has been designed for objective functions having long simulation times, thus offering the possibility of evaluating each function in parallel. The parallel code of Optimus has been created from scratch, so any potential similarity with Paradiseo has occurred by chance.

2.4 Validation tests

Now that the first version of the optimization library has been created, it is necessary to test the correctness of the implemented algorithms. An exhaustive check of every feature was carried out by the author. Most of the tests were more related to programming issues and are not shown in this Doctoral Thesis. The validation tests related to the conceptual development of the library are presented in 2 steps: i) based on mathematical functions contained in common optimization test suites, and ii) based on real-world simulations belonging to the field of Computational Fluid Dynamics & Heat Transfer (CFD & HT).

Validation tests measure the accuracy and performance of the optimization library. Accuracy is calculated by comparing the known optimal solution of the benchmark case with the solution provided by the optimization algorithm. The performance may be obtained in several ways: measuring the number of objective function evaluations, measuring simulation time, etc. At this development stage, accuracy is considered important and little attention is paid to performance, i.e. no comparison of different methods and parametrizations of the optimization algorithm is carried out. The reason is that no effort was made so far to optimize the performance of the code, so some measurements are shown but just for information purposes. The unique goal is to provide a few results which prove the correctness of the library's implementation.

2.4.1 Benchmark mathematical functions

In Chapter 1 several reputed test suites were mentioned, together with a selection of some functions for this Doctoral Thesis. On one hand, the Rosenbrock, Rastrigin, Schwefel and Griewank functions were selected for testing the single-objective optimization algorithms. On the other hand, Kursawe's function (MOP4) and Zitzler-Deb-Thiele's function N. 6 (ZDT6) were the preferred functions for testing multi-objective optimization algorithms. Since the evaluation time of these mathematical functions is extremely short, all simulations were run sequentially in 1 processor of the supercomputer and hence the parallel features of Optimus were not used.

Single-objective tests

The selected test functions were simulated with 2, 4 and 6 optimization variables. The stopping criterion for the optimizer was to reach a target objective value of 1e-9, being the real optimal value equal to 0 for all functions. Each function was optimized twice: once using exclusively the genetic algorithm, and once using the hybrid algorithm composed by the genetic algorithm and the local search method in Trilinos/Moocho package. Results are shown in Table 2.2 and include the obtained optimal objective value, the number of function evaluations carried out and the required wall clock time for each optimization.

Due to the decimal precision used in the implementation of Schwefel's function, the optimizer was unable to reach the required accuracy. Nevertheless, the obtained solutions are considered to be satisfactory.

Regarding the Griewank's test function, note that the results obtained by the hybrid optimizer are not shown. The reason is that it was unable to find the global optimum of

the function with the required accuracy. This fact is explained by the noisy nature of the function's fitness landscape. Note also that the optimization variables were bounded in the range $[-50,50]$.

Test function	Benchmark optimum	Genetic algorithm			Hybrid algorithm		
		Optimum	No. function evaluations	Time (seconds)	Optimum	No. function evaluations	Time (seconds)
Rosenbrock_ 2vars	0	6.6658e-10	50048	6.7528	8.8794e-12	169	0.0172
Rosenbrock_ 4vars	0	9.9656e-10	59920	7.1141	2.0650e-11	1031	0.1171
Rosenbrock_ 6vars	0	9.8136e-10	95270	8.9573	2.5048e-11	4120	0.4481
Rastrigin_ 2vars	0	4.6314e-10	1400	0.2051	0	332	0.0550
Rastrigin_ 4vars	0	6.9260e-10	5360	0.6044	0	2254	0.2403
Rastrigin_ 6vars	0	7.8748e-10	11680	0.9840	0	6859	0.6079
Schwefel_ 2vars	0	2.5455e-05	1376	0.1951	2.5455e-05	215	0.0668
Schwefel_ 4vars	0	5.0910e-05	5450	0.6062	5.0910e-05	1823	0.1825
Schwefel_ 6vars	0	7.6365e-05	14860	1.1752	7.6365e-05	3519	0.3304
Griewank_ 2vars	0	2.2690e-10	1184	0.1669	-	-	-
Griewank_ 4vars	0	7.9315e-10	21220	1.4272	-	-	-
Griewank_ 6vars	0	9.0648e-10	31900	3.0057	-	-	-

Table 2.2: Results of the single-objective test functions.

Multi-objective tests

Multi-objective test functions MOP4 and ZDT6 are shown respectively in Fig. 2.4 and Fig. 2.5. The true optimal Pareto front (named Benchmark) and the Pareto front obtained by the optimizer (named Optimus) are represented in both figures. 50100 objective function evaluations were needed in order to build each one of the Pareto fronts corresponding to MOP4 and ZDT6. Performing those evaluations took 94.88 seconds in the first case and 28.45 seconds in the latter. It is considered that 1 evaluation involves calculating Objective 1 and Objective 2 in both tests. It can be seen that the accuracy and diversity of the Pareto front found by Optimus is acceptable, although better results could be found by assigning either more time or computational resources.

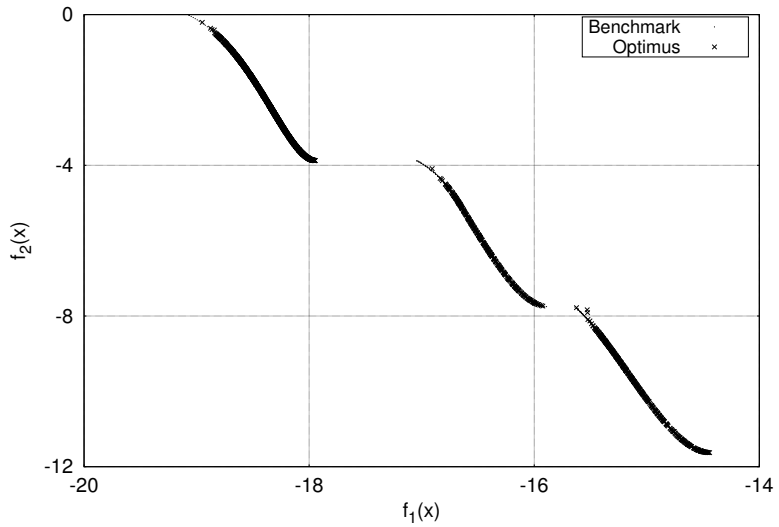


Figure 2.4: Pareto Front obtained by Optimus for the Kursawe's function (MOP4).

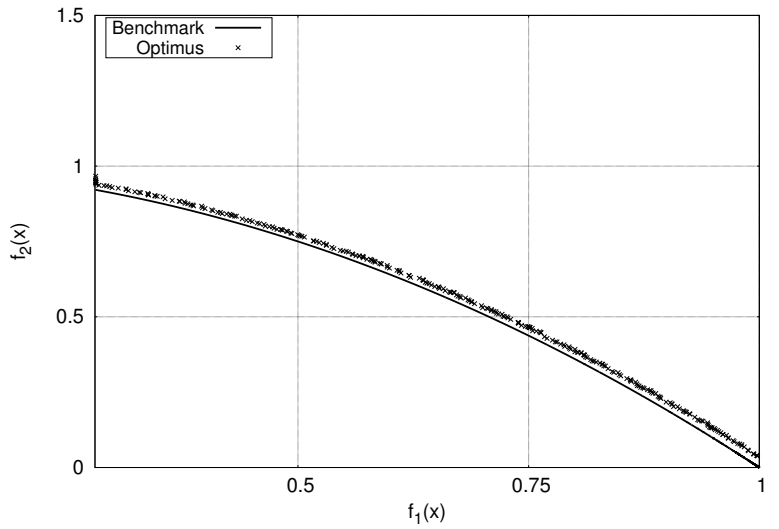


Figure 2.5: Pareto Front obtained by Optimus for the Zitzler-Deb-Thiele's function N.6 (ZDT6).

2.4.2 CFD & HT tests

Two different real-world cases from the CFD & HT field are presented next. The first case consists on the energy labelling of a fridge, which is based on a real industrial problem solved at CTTC [25]. The second case is a CFD simulation of an incompressible fluid circulating through a rectangular pipe, where the optimal geometry of the pipe is searched.

Optimization of the energy efficiency index of a fridge

The energy efficiency index of a fridge is calculated according to 2 main characteristics: the energy consumption of the fridge and its useful (internal) volume. The lower the efficiency index, the fridge's energy labelling is better and its price in the market higher. However, the optimal relation between the energy consumption and the useful volume is not trivial. The reason is that having thinner walls, i.e. greater internal volume, increases the heat losses due to lack of thermal isolation. The task of the optimization algorithm consists on finding the optimal balance between these two characteristics for a set of 3 problems of industrial interest.

The first step has been to create a simplified mathematical model of the fridge, which may be solved both analytically and using more advanced computational tools. In this model, the fridge is composed by N walls. Each wall has the following characteristics:

A_i : *area*
k_i : *thermal conductivity*
d_i : *thickness*
ΔT_i : *temperature gradient*

The total conduction heat loss through the walls is defined by the following expression:

$$Q = \sum_{i=1}^{N} k_i A_i \Delta T_i \,/\, d_i \tag{2.1}$$

Two volumes are defined: the external volume of the fridge (V_o) and the internal or useful volume (V). V_o has a constant fixed value, whereas the internal volume is defined as:

$$V = V_o - \sum_{i=1}^{N} A_i d_i \tag{2.2}$$

The energy efficiency index (E_i) is defined according to the laws in force in 2013 (see Fig. 2.6):

$$E_i = E_a / E_{st} \tag{2.3}$$

where the term E_a is related to the energy consumption of the appliance and E_{st} is related to the internal volume.

On one side, E_a is defined as follows:

$$E_a = 100 \cdot k1 \cdot Q \tag{2.4}$$

where

$$k1 = \frac{365 \cdot 24}{1000 \cdot COP} \tag{2.5}$$

being COP the Coefficient of Performance. On the other side, E_{st} is defined as in Eq. 2.6 for A+ and A++ energy labels and as in Eq. 2.7 otherwise:

$$E_{st} = M \cdot (AV) + N + CH \tag{2.6}$$

$$E_{st} = M \cdot (AV) + N \tag{2.7}$$

where M and N depend on the appliance class and CH is a correction factor (see Fig. 2.6). AV is defined as in Eq. 2.8 for A+ and A++ energy labels and as in Eq. 2.9 otherwise:

$$AV = \sum \left(V_c \frac{(25 - T_c)}{20} \cdot FF \cdot CC \cdot BI \right) \tag{2.8}$$

$$AV = V_R + \Omega V_F \tag{2.9}$$

where T_c is the compartment temperature, V_c is the compartment storage volume, $FF/CC/BI$ are several correction factors tabulated in Fig. 2.6, V_R is the fresh food storage compartment volume, V_F is the frozen food storage compartment value and Ω is dependent on the appliance class (see Fig. 2.6).

Two implementations of the described mathematical model were carried out. The first one consists on just writing the formulation using C++ functions. The second one consists on building a multi-physics model of the fridge using CTTC's in-house multi-physics software, called NEST [19]. The aim of the latter implementation is to

test the coupling between Optimus and NEST, which was one of the design criteria of Optimus.

The multi-physics model has a main system, called Fridge, at the top level. This system is composed by 2 subsystems, namely the refrigerator compartment and the freezer compartment. The refrigerator compartment subsystem is composed by 5 walls (named as refrigerator right side, refrigerator left side, refrigerator rear, refrigerator top and refrigerator door), whereas the freezer compartment subsystem is composed by 9 walls (named as bottom, compressor zone right side, compressor zone left side, compressor zone vertical side, compressor zone horizontal size, freezer right side, freezer left side, freezer rear, freezer door). Thus, the overall number of walls is 14 and the total volume and heat losses of the fridge are the sum of the volumes and heat losses of both compartments.

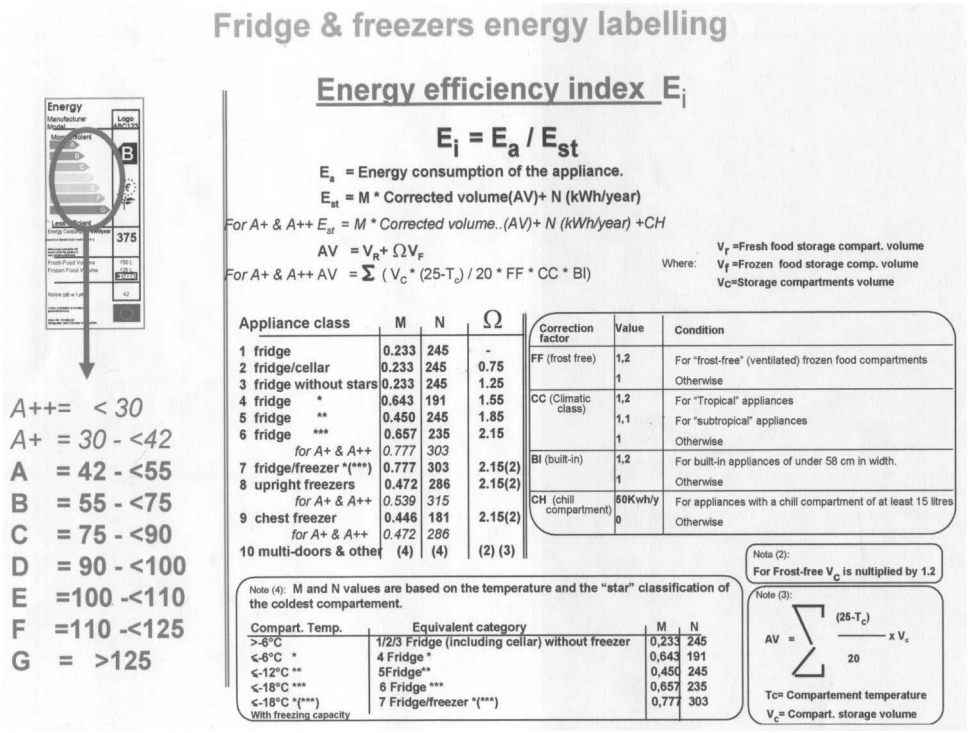


Figure 2.6: Tables for calculating fridges' and freezers' energy efficiency index (E_i) according to the laws in force in 2013.

3 different optimization problems are presented next. The ΔT and the area of each wall are known input parameters, as well as the coefficients tabulated in Fig. 2.6. The aim of the optimizations is always to find a set of wall thickness values corresponding to the optimum design according to an objective function. The objective function is different in all 3 problems.

The benchmark results have been obtained by using the analytical method of Lagrange multipliers applied to the simplest implementation of the objective function, i.e. the formulation written using C++ functions. On the other hand, the multi-physics model has been the objective function evaluated by Optimus.

Since the proposed optimization problems are extracted from a real world industrial project, real geometrical data, operation points and other confidential information is not provided. This is why every wall thickness value of each optimization problem has been normalized using the biggest thickness in the benchmark results of that problem.

Problem 1: Find the set of thickness values which minimizes the energy efficiency index (E_i) of the fridge.

$$min\ f(\mathbf{d}) = E_i(\mathbf{d})$$

$$s.t. \quad E_i > 0$$

$$and \quad d_i > 0 \ \ for\ i = 1,...,14$$

where $\mathbf{d}$ is a vector containing the 14 d_i wall thickness values.

The obtained optimal sets of thickness values are shown in Table 2.3, whereas the associated heat losses, volumes and energy efficiency indexes are shown in Table 2.4. It took 2464 objective function evaluations to the genetic algorithm to reach those results, whereas it took 584 evaluations to the hybrid algorithm.

Wall	Benchmark thickness	Genetic algorithm's thickness	Hybrid algorithm's thickness
Bottom	0.922	0.926	0.926
Compr. zone right side	0.922	0.926	0.926
Compr. zone left side	0.922	0.926	0.926
Compr. zone vertical side	0.972	0.976	0.976
Compr. zone horiz. side	1.000	1.005	1.005
Freezer right side	0.922	0.926	0.926
Freezer left side	0.922	0.926	0.926
Freezer rear	1.000	1.005	1.005
Freezer door	0.922	0.926	0.926
Refrigerator right side	0.615	0.617	0.617
Refrigerator left side	0.615	0.617	0.617
Refrigerator rear	0.727	0.731	0.731
Refrigerator top	0.615	0.617	0.617
Refrigerator door	0.615	0.617	0.617

Table 2.3: Optimal sets of thickness values for Problem 1.

	Benchmark value	Genetic algorithm		Hybrid algorithm	
		Value	Rel. error	Value	Rel. error
Q [W]	32.25	32.11	0.4%	32.11	0.4%
V insulation [liters]	401.20	402.90	0.4%	402.90	0.4%
V refrigerator [liters]	146.50	145.30	0.8%	145.30	0.8%
V freezer [liters]	58.20	57.80	0.7%	57.80	0.7%
E_i	30.68	30.69	0.03%	30.69	0.03%

Table 2.4: Energy labelling parameters associated to the optimal sets of thickness values found for Problem 1. The values found by the genetic and the hybrid algorithms are compared with the benchmark, being the relative error measured.

Problem 2: Find the set of thickness values which minimizes the energy efficiency index (E_i) of the fridge being the internal (useful) volume fixed.

$$min\ f(\mathbf{d}) = Q(\mathbf{d})$$

$$s.t. \quad d_N > 0$$

$$and \quad d_i > 0 \ \ for\ i = 1,...,13$$

where $\mathbf{d}$ is a vector containing 13 d_i wall thickness values (all except the N-th thickness value).

Since the internal volume of the fridge is a known parameter, the N-th thickness has been extracted from the set of optimization variables and is calculated as a function of the rest of geometrical data:

$$d_N = \frac{V_{insul} - \sum d_i A_i}{A_N} \tag{2.10}$$

where

$$V_{insul} = V_o - V \tag{2.11}$$

It was decided to define the objective function as minimizing the overall heat loss Q, since minimizing Q implies minimizing E_i for a fixed internal volume of the fridge.

The obtained optimal sets of thickness values are shown in Table 2.5, whereas the associated heat losses, volumes and energy efficiency indexes are shown in Table 2.6. It took 9010 objective function evaluations to the genetic algorithm to reach those results, whereas it took 988 evaluations to the hybrid algorithm. Note that the genetic algorithm was stopped manually when stagnation of the best individual was detected.

Wall	Benchmark thickness	Genetic algorithm's thickness	Hybrid algorithm's thickness
Bottom	0.921	0.905	0.921
Compr. zone right side	0.921	1.034	0.921
Compr. zone left side	0.921	0.979	0.921
Compr. zone vertical side	0.972	0.962	0.972
Compr. zone horiz. side	1.000	0.995	1.000
Freezer right side	0.921	0.920	0.921
Freezer left side	0.921	0.912	0.921
Freezer rear	1.000	0.991	1.000
Freezer door	0.921	0.914	0.921
Refrigerator right side	0.615	0.615	0.614
Refrigerator left side	0.615	0.615	0.614
Refrigerator rear	0.727	0.723	0.727
Refrigerator top	0.615	0.621	0.614
Refrigerator door	0.615	0.615	0.614

Table 2.5: Optimal sets of thickness values for Problem 2.

	Benchmark value	Genetic algorithm		Hybrid algorithm	
		Value	Rel. error	Value	Rel. error
Q [W]	46.59	46.62	0.06%	46.60	0.02%
V insulation [liters]	277.60	277.60	0%	277.60	0%
V refrigerator [liters]	234.90	235.00	0.04%	235.00	0.04%
V freezer [liters]	93.40	93.40	0%	93.40	0%
$\mathbf{E_i}$	33.97	33.98	0.03%	33.97	0%

Table 2.6: Energy labelling parameters associated to the optimal sets of thickness values found for Problem 2. The values found by the genetic and the hybrid algorithms are compared with the benchmark, being the relative error measured.

Problem 3: Find the set of thickness values which maximizes the internal (useful) volume of the fridge being the energy efficiency index (E_i) fixed.

$$
\begin{aligned}
& min\ f(\mathbf{d}) = V_{insul}(\mathbf{d}) \\
& s.t. \quad V_{insul} < V_o \\
& \qquad E_i = E_{i,fixed} \\
& and \quad d_i > 0 \ \ for\ i = 1,...,14
\end{aligned}
$$

where $E_{i,fixed} = 34.7$, $\mathbf{d}$ is a vector containing the 14 d_i wall thickness values and the insulation volume is calculated as in Eq. 2.11.

The difficulty of solving this optimization problem with Optimus lies on the fact that the implementation of equality constraint handling methods is not finished for the genetic algorithm. As a consequence, there is no straightforward way to treat this kind of constraints to date. Therefore, another approach to the same optimization problem is proposed:

$$
\begin{aligned}
& min\ f(\mathbf{d}) = w_1 \cdot V_{insul}(\mathbf{d}) \ + \ w_2 \cdot (E_i - E_{i,fixed})^2 \\
& s.t. \quad V_{insul} < V_o \\
& and \quad d_i > 0 \ \ for\ i = 1,...,14
\end{aligned}
$$

where $E_{i,fixed} = 34.7$, $w_1 = 1$, $w_2 = 1e-4$, $\mathbf{d}$ is a vector containing the 14 d_i wall thickness values and the insulation volume is calculated as in Eq. 2.11.

It can be seen that the objective function is now composed by two weighted terms. The first term is related to the insulation volume, whereas the second term represents the quadratic error of E_i compared to $E_{i,fixed}$. This second term guarantees the fulfillment of the equality constraint, provided that appropriate weight values are defined. The proposed values for w_1 and w_2 were chosen because this allows calculating both E_i and the insulation volume (in liters) with at least 1 decimal precision.

The obtained optimal sets of thickness values are shown in Table 2.7, whereas the associated heat losses, volumes and energy efficiency indexes are shown in Table 2.8. It took 50024 objective function evaluations to the genetic algorithm to reach those results, whereas it took 4985 evaluations to the hybrid algorithm. Note that the genetic

algorithm was stopped manually when stagnation of the best individual was detected.

Wall	**Benchmark thickness**	**Genetic algorithm's thickness**	**Hybrid algorithm's thickness**
Bottom	0.922	0.898	0.920
Compr. zone right side	0.922	1.017	0.920
Compr. zone left side	0.922	0.890	0.920
Compr. zone vertical side	0.972	1.037	0.971
Compr. zone horiz. side	1.000	1.011	0.999
Freezer right side	0.922	0.931	0.920
Freezer left side	0.922	0.913	0.920
Freezer rear	1.000	0.990	0.999
Freezer door	0.922	0.909	0.920
Refrigerator right side	0.614	0.612	0.613
Refrigerator left side	0.614	0.607	0.613
Refrigerator rear	0.727	0.723	0.726
Refrigerator top	0.614	0.623	0.613
Refrigerator door	0.614	0.619	0.613

Table 2.7: Optimal sets of thickness values for Problem 3.

	Benchmark value	**Genetic algorithm**		**Hybrid algorithm**	
		Value	**Rel. error**	**Value**	**Rel. error**
Q [W]	48.66	48.62	0.08%	48.66	0%
V insulation [liters]	265.80	266.20	0.2%	265.90	0.04%
V refrigerator [liters]	243.10	243.10	0%	243.40	0.1%
V freezer [liters]	96.70	96.70	0%	96.80	0.1%
E_i	34.70	34.69	0.03%	34.70	0%

Table 2.8: Energy labelling parameters associated to the optimal sets of thickness values found for Problem 3. The values found by the genetic and the hybrid algorithms are compared with the benchmark, being the relative error measured.

Two additional studies were carried out using the multi-objective optimization features implemented in Optimus. The first one shows the variation of the energy efficiency index E_i as a function of the insulation volume V_{insul} (see Fig. 2.7). The second one shows the variation of the total heat loss Q_{loss} as a function of the insulation volume V_{insul} (see Fig. 2.8).

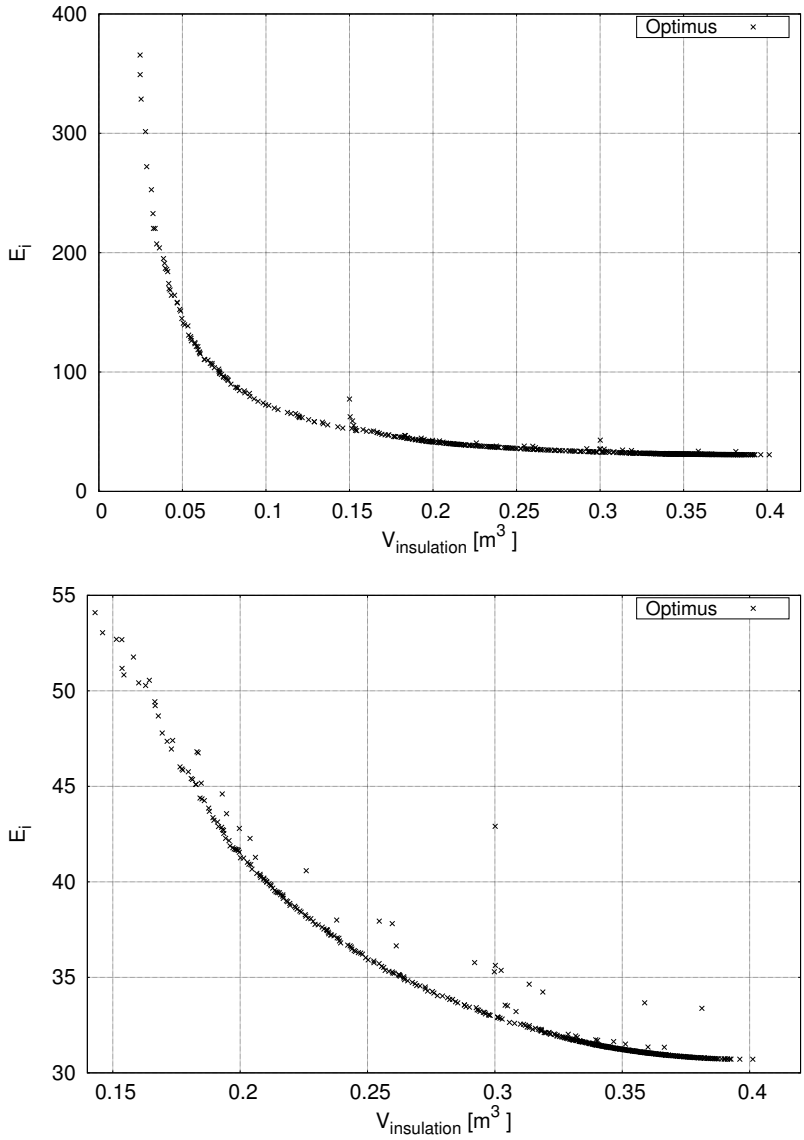


Figure 2.7: Variation of the best attainable energy efficiency index (E_i) as a function of the insulation volume (V_{insul}). Upper graph: $V_{insulation}$ is represented in the whole range. Lower graph: $V_{insulation}$ is represented in a smaller range, being every point extracted from the upper graph.

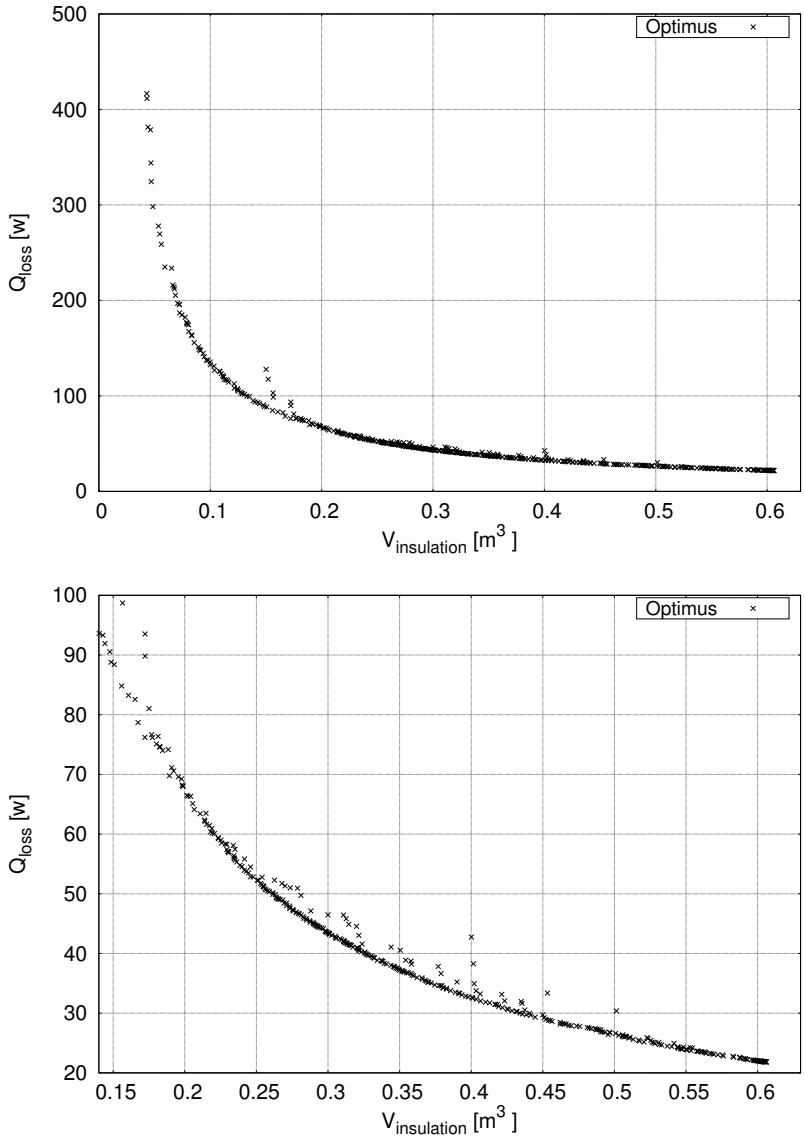


Figure 2.8: Variation of the minimal attainable total heat loss (Q_{loss}) as a function of the insulation volume (V_{insul}). Upper graph: $V_{insulation}$ is represented in the whole range. Lower graph: $V_{insulation}$ is represented in a smaller range, being every point extracted from the upper graph.

Both graphs were obtained by solving 2-objective optimization problems in which both objectives had to be minimized by assigning appropriate values to the set of 14 thicknesses. The obtained optimal Pareto fronts are shown in the graphs, which represent the best attainable values of E_i and Q_{loss} for each value given to V_{insul}. Note that a few points dominated by other points (according to Pareto's criterion) are shown. This happened because several simulations for different V_{insul} ranges were first run and the obtained results were finally assembled giving rise to the Pareto fronts in Fig. 2.7 and Fig. 2.8, being the dominated points not filtered after assembling the initial results.

For concluding this section focused on the energy optimization of a fridge, it can be said that the Optimus library solved successfully all presented industrial optimization problems. Moreover, it has been proved that the coupling between Optimus and CTTC's multi-physics software NEST works fine and is comfortable to use.

Surface morphing optimization problem

Surface morphing is one of the applications for which an optimization algorithm may be used. The need to find the optimal shape of a component which reduces aerodynamic drag, for instance, is a common problem in the field of aerodynamics. Finding such geometry is a difficult task and carrying out a parametric study may involve a large amount of computational time. The finer the mesh used for modeling the geometry, the more computational time is required. The most interesting aspects of facing such a problem with Optimus are twofold: i) it needs to be coupled to the TermoFluids package, the in-house CFD software at CTTC, and ii) the parallelization algorithm of Optimus can be tested.

A two dimensional surface morphing problem is proposed next, based on the case of a duct with variable indentation taken from [26]. The geometric parameters of the mathematical model are shown in Fig. 2.9 and in the following equation,

$$y(x) = \begin{cases} h & |x| \in [0, x_1] \\ 0.5h\ [1 - tanh(\ a(|x| - x_2)\ )] & |x| \in [x_1, x_3] \\ 0 & |x| > x_3 \end{cases} \tag{2.12}$$

where $a = 4.14$, $x_1 = 4b$, $x_3 = 6.5b$ and $x_2 = 0.5(x_1 + x_3)$. h is the indentation, bounded in $[0, 0.38 \cdot b]$. b is a parameter representing the height of the duct, and for the present test case it is defined to be $b = 1.0$. The flow at the initial state is assumed to be completely

developed with velocity

$$\mathbf{v} = (6y(1-y), 0, 0)^T \tag{2.13}$$

The inlet flow is set to be constant according to Eq. 2.14

$$U = \int_0^b 6y(1-y)\,dy = 1 \tag{2.14}$$

A pressure-based condition is applied at the outlet boundary. The Reynolds number is fixed to 507 and the Prandtl number to 0.71. As the indentation is varied, the velocity contour maps are altered as it is shown in Fig. 2.10.

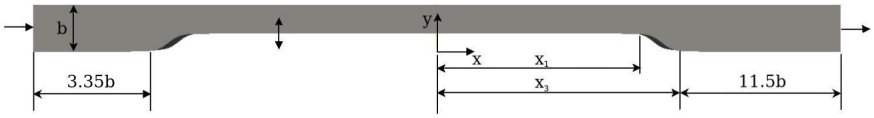


Figure 2.9: Geometry of the test case based on a duct with a moving indentation, with $b = 1$ (extracted from [26]).

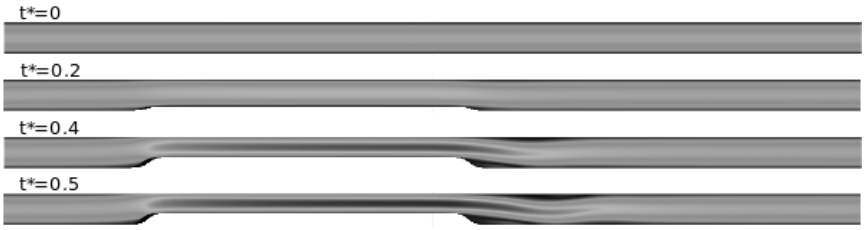


Figure 2.10: Velocity contour maps for different indentation values (extracted from [26]).

The proposed optimization problem makes use of this mathematical model and is stated as follows:

Problem 4: Find the optimal indentation of the pipe which minimizes the mean horizontal velocity of the fluid at x=0.

$min\ f(h) = U_{mean}(h) \quad at \quad x = 0$

where $0 \leq h \leq 0.38$

The problem has one single optimization variable and the mean horizontal velocity at x=0 can be calculated as follows:

$$U_{mean} = \frac{1}{b-h} \int_h^b 6y(1-y)dy \tag{2.15}$$

Due to the simplicity of the problem, it is known that the optimal indentation value is h=0.

The implementation has been carried out in the following way. A mesh for the rectangular pipe with no indentation has been first done by means of ICEMCFD [27]. When a new mesh is needed for a certain indentation value, a parallel radial basis function method for unstructured dynamic meshes [26] is used in order to displace step by step the nodes of the base mesh until the desired indented geometry is reached. Once the required mesh is constructed, the flow is solved by means of an explicit CFD solver from the TermoFluids package. It is made sure that the simulation time is enough to reach a steady turbulent state before the mean horizontal velocity at $x = 0$ is calculated. The cross section of the pipe has been discretized in 100 equally sized control volumes for U_{mean}'s calculation.

Two optimizations have been carried out using the genetic algorithm and the hybrid algorithm respectively. The same continuation criterion was selected in both cases, i.e. to stop the optimization when the best individual was not improved after a certain number of generations. On one hand, it took 925 function evaluations to the genetic algorithm to reach an objective value of 0.99036. The indentation corresponding to this result was 1.79338e-06. On the other hand, it took 676 function evaluations to the hybrid algorithm to reach an objective value of 0.990356, being the indentation corresponding to this result 2.94003e-05.

The theoretical optimal objective value is 1, which is obtained for an indentation value of $h = 0$. It can be seen that the obtained results are very close, being deviations attributed to the inaccurate discretization of the integral used for calculating U_{mean}.

The obtained results are considered satisfactory, mainly because a successful Optimus-TermoFluids coupling was achieved and because the correctness of the parallel task manager's implementation was proved. Good results were also provided for the optimization problem, although this was a secondary concern due to its simplicity.

2.5 Conclusions

A new optimization library named Optimus has been implemented in this second chapter. The first step has been the definition of general and specific requirements of the library. The general requirements are based on common design concepts of a framework, whereas the specific requirements are conditioned by the research field and the infrastructure available at the research institution (CTTC). Thus, a brief description of the available computing facilities has been included.

A state of the art study on already available optimization libraries has been conducted next, considering both free open-source software and proprietary commercial software. After comparing several packages, it has been decided to base the development of Optimus in two of them: Paradiseo and Trilinos/Moocho, which are both free open-source packages. Paradiseo has been chosen because of its implementation of genetic algorithms, whereas the availability of gradient-methods suitable for local search has been the appealing feature of Trilinos/Moocho.

The main features of Optimus are explained in the subsequent section. The development strategy for the new library is first introduced. The implemented algorithms are described next, grouped under the following sections: definition of the optimization problem, single-objective vs. multi-objective optimization, genetic operators, hybrid methods, continuation criteria, statistics, parallelization, user interface and other features. Finally, a brief comparison between Optimus and Paradiseo is carried out with the aim of remarking the improvements available in the new library.

The last section of this chapter contains the results obtained in two types of validation tests: benchmark mathematical functions and specific tests from the field of Computational Fluid Dynamics and Heat Transfer (CFD & HT). On one hand, the simulated mathematical functions have been extracted from common optimization test suites designed for both single-objective and multi-objective cases. On the other hand, CTTC has a vast experience on CFD & HT simulations and two cases have been selected: the optimization of the energy efficiency of a fridge, and the optimization of the geometry of a pipe. Optimus has been able to successfully carry out all tests, which proves that the library is ready to be used for solving real-world optimization problems.

Acknowledgments

This work has been financially supported by a FPU Doctoral Grant (Formación de Profesorado Universitario) awarded by the Ministerio de Educación, Cultura y Deporte, Spain (FPU12/06265) and by Termo Fluids S.L. The author thankfully acknowledges these institutions.

References

[1] S. Cahon, N. Melab, and E. G. Talbi. Building with ParadisEO reusable parallel and distributed evolutionary algorithms. *Parallel Computing*, 30(5):677–697, 2004.

[2] O. Lehmkuhl, C. D. Perez-Segarra, R. Borrell, M. Soria, and A. Oliva. TERMOFLUIDS: A new parallel unstructured CFD code for the simulation of turbulent industrial problems on low cost PC cluster. In *Parallel Computational Fluid Dynamics 2007*, pages 275–282. Springer, 2009.

[3] Message Passing Interface Forum. *MPI: A Message-Passing Interface standard, version 3.1.* High Performance Computing Center Stuttgart (HLRS), 2015.

[4] A. Liefooghe, M. Basseur, L. Jourdan, and E. G. Talbi. ParadisEO-MOEO: A framework for evolutionary multi-objective optimization. In *International Conference on Evolutionary Multi-Criterion Optimization (EMO 2007)*, volume 4403, pages 386–400, Matsushima, Japan, 2007. Springer.

[5] E. Alba and J. M. Troya. A survey of parallel distributed genetic algorithms. *Complexity*, 4(4):31–52, 1999.

[6] M. Keijzer, J. J. Merelo, G. Romero, and M. Schoenauer. Evolving objects: A general purpose evolutionary computation library. In *Proceedings of the 5th International Conference on Artificial Evolution (EA'01)*, pages 231–242, Le Creusot, France, 2001. Springer.

[7] Paradiseo, a software framework for metaheuristics. http://paradiseo.gforge.inria.fr.

[8] M. Wall. GAlib: A C++ library of genetic algorithm components. *Mechanical Engineering Department, Massachusetts Institute of Technology*, 87:54, 1996.

[9] S. Adcock. GAUL, the Genetic Algorithm Utility Library. http://gaul.sourceforge.net, 2009.

[10] R. Berlich, S. Gabriel, A. Garcia, and M. Kunze. Distributed parametric optimization with the Geneva library. In *Data Driven e-Science*, pages 303–314. Springer, 2011.

[11] B. M. Adams, L. E. Bauman, W. J. Bohnhoff, K. R. Dalbey, M. S. Ebeida, J. P. Eddy, M. S. Eldred, P. D. Hough, K. T. Hu, J. D. Jakeman, J. A. Stephens, L. P. Swiler, D. M. Vigil, and T. M. Wildey. Dakota, a multilevel parallel object-oriented framework for design optimization, parameter estimation, uncertainty quantification, and sensitivity analysis: version 6.0 user's manual. Sandia Technical Report SAND2014-4633, July 2014. Updated November 2015 (Version 6.3).

[12] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, and E. T. Phipps. An overview of the Trilinos project. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):397–423, 2005.

[13] IOSO NM Version 1.0, User's Guide. IOSO Technology Center, Moscow, Russia, 2003.

[14] MATLAB Release 2016a. The MathWorks, Inc., Natick, Massachusetts, United States, 2016.

[15] LINGO 16.0. Lindo Systems Inc., Chicago, Illinois, United States.

[16] H. P. Schwefel. *Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie*, volume 1. Birkhäuser, Basel, Switzerland, 1977.

[17] H. P. Schwefel. Internal Report of KFA Juelich, KFA-STE-IB-3/80, 1980.

[18] G. Rudolph. Globale Optimierung mit parallelen Evolutionsstrategien. Diploma thesis, University of Dortmund, 1990.

[19] R. M. Damle, O. Lehmkuhl, G. Colomer, and I. Rodríguez. Energy simulation of buildings with a modular object-oriented tool. In *ISES Solar World Congress 2011*, pages 1–11, 2011.

[20] A. Liefooghe, L. Jourdan, and E. G. Talbi. A unified model for evolutionary multiobjective optimization and its implementation in a general purpose software framework: ParadisEO-MOEO. Research Report RR-6906, INRIA, 2009.

[21] D. E. Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley, 1989.

[22] N. Marco and S. Lanteri. A two-level parallelization strategy for genetic algorithms applied to optimum shape design. *Parallel Computing*, 26(4):377–397, 2000.

[23] M. S. Eldred, W. E. Hart, B. D. Schimel, and B. G. van Bloemen Waanders. Multilevel parallelism for optimization on MP computers: Theory and experiment. In *Proc. 8th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, number AIAA-2000-4818, Long Beach, CA*, volume 292, pages 294–296, 2000.

[24] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[25] Modular domestic refrigeration systems with high energy efficiency (KERS). Research Project F00299, Ref. IPT-020000-2010-30, INNPACTO (Spanish Government), Company: Fagor Electrodomésticos, Funding: 439552 Euros, Period: 2010-2013.

[26] O. Estruch, O. Lehmkuhl, R. Borrell, C. D. Pérez-Segarra, and A. Oliva. A parallel radial basis function interpolation method for unstructured dynamic meshes. *Computers & Fluids*, 80:44–54, 2013.

[27] Inc. ANSYS. ANSYS® ICEM CFD, Release 16.2.

3

Load balancing methods for parallel optimization algorithms

Abstract. The aim of this third chapter is to evaluate the impact of several load balancing methods on the parallel efficiency of optimization algorithms. After carrying out a state of the art study of the available techniques, new load balancing algorithms are proposed and tested by means of an exhaustive theoretical case study. Finally, they are applied to the optimization of a real-world engineering model consisting on the refrigeration of a power electronic device.

3.1 Introduction

A state of the art study on the parallelization of genetic and other population-based optimization algorithms was carried out in section 1.7. Regarding hardware parallelization, finding an appropriate balance of the computational load was identified as the key point to avoid the degradation of the optimization algorithm's scalability. This applies in case the average evaluation time of the batch of individuals is greater than the average time required by the optimizer for performing inter-processor communications. In the case of a synchronous global parallel genetic algorithm (i.e. an algorithm that synchronizes all processors after each generation, stopping and waiting to receive the fitness values for all the population before starting the next generation) or any other optimization algorithm which synchronizes all processors after the evaluation of every batch of individuals, the main potential causes of computational load imbalance are the following:

- Inappropriate ratio (no. individuals / no. processor groups): The processor groups with fewer individuals to evaluate will remain idle while the rest of the groups finish their pending evaluations. This is likely to happen when a hybrid optimization method is used, e.g. genetic algorithm + gradient method, because the batches of individuals to be solved by each constituent algorithm have very dissimilar sizes.
- Heterogeneous parallel computer systems: Hardware heterogeneity translates into non-homogeneous objective evaluation times and the consequent loss of parallel performance in the case of optimization algorithms.
- Heterogeneous objective function evaluation time: This phenomenon happens when the evaluation cost of individuals is dependent on the optimization variables, i.e. the input variables of the objective functions, and is not unusual in heat transfer and nonlinear mechanics applications.

The greater the number of processors used, the greater the degradation of the scalability caused by computational load imbalance. Taking into account that we stand on the threshold of the exascale era, the need of codes that ensure an efficient use of computational resources is becoming crucial. However, and as stated in section 1.7, scarce research has been found related to hardware parallelization in optimization algorithms.

It may seem strange at first glance to find a Doctoral Thesis dedicated to the parallelization of optimization algorithms hosted by a research group in the field of

Computational Fluid Dynamics & Heat Transfer (CFD & HT). Nonetheless, the motivation of this work may be understood more easily when taking into account that CFD & HT is a field at which physics, applied mathematics and computer science join together. CFD simulations are characterized by their high computational cost. Consequently, an unbalanced load distribution among processors can easily lead to a considerable waste of computational time.

An illustrative example of a CFD simulation that requires advanced load balancing algorithms is described in [1]. The behavior of a fluid in a certain domain is discretized using a mesh. If higher resolution is needed at some locations during the simulation, the mesh is dynamically altered by means of an adaptation technique. These local mesh refinements, however, increase the number of cells. The consequent adverse effect on the computational load balance in case an optimization is being run is twofold. First, there is a direct relationship between the number of cells in a model and its computational cost. As the number of cells increases, so does the evaluation time of the individual. Therefore, a batch of individuals with meshes having a different number of cells is heterogeneous regarding evaluation times. Second, the mesh of each individual is partitioned among the processors in charge of its evaluation by means of domain decomposition techniques. In the case of having a mesh with a constant number of cells, it is enough to perform partitioning before evaluating the individuals. Nonetheless, the use of adaptation techniques during a parallel simulation leads to computational load change throughout the domain, making the initial load balancing ineffective. If one wants to maintain a proper parallel efficiency, a dynamic load balancing scheme has to be introduced. This example shows that the use of sophisticated techniques for optimizing CFD & HT models may involve important computational load imbalance problems.

The load balancing methods developed in this chapter have been applied to the genetic algorithm. The reason is that this is the most widely used optimization heuristic, so the impact of the research is expected to be maximized. Moreover, the genetic algorithm is the only global search technique implemented in Optimus (see Chapter 2). Anyway, it must be borne in mind that the proposed algorithms and the reached conclusions are extendable to any other optimization method evaluating batches of individuals and synchronizing all processors after the evaluation of each batch.

3.2 Approach to the load balancing problem

3.2.1 Definitions

Load balancing problem
The load balancing problem by means of task scheduling consists on assigning n tasks to p resources in such a way that the total completion time of the task set (makespan) is minimized [2]. The minimum makespan comes true when the computational load is well balanced, i.e. when every resource is occupied and efficiently used. This balancing process is an NP-complete problem and may be carried out statically, i.e. before the set of tasks begins to be processed, or dynamically, i.e. as simulation progresses and computational requirements become clearer.

Nature of tasks
Two concepts must be introduced before addressing the load balancing problem: divisible load theory and job flexibility.

Divisible load theory is a methodology involving the linear and continuous modeling of partitionable computation and communication loads for parallel processing [3]. This methodology seeks to divide the total load into smaller fractions that can be processed independently. Once every fraction has been processed, the partial solutions can be consolidated to construct the complete solution to the problem. The partitioning depends on the divisibility of the load, which is the property determining whether a load can be decomposed into a set of smaller loads or not. The following classification is proposed in [3]:

- ***Indivisible loads:*** Loads that cannot be subdivided and are independent from any other job, thus not having any precedence relations. They have to be processed in their entirety in a single processor.
- ***Divisible loads:*** Loads that can be subdivided into smaller fractions.
  - ***Modularly divisible:*** These loads can be subdivided into smaller modules based on some characteristic of the loads or the system in charge of processing them. The processing of a load is complete when all its modules are processed. Moreover, the processing of the modules may be subject to precedence relations.

  - ***Arbitrarily divisible:*** These loads can be arbitrarily partitioned into any number of load fractions, and all elements in the load demand an identical type of processing. These load fractions may have precedence relations.

In the case of arbitrarily divisible parallel applications (or jobs), the internal design of the job may ease or constrain the partitioning into smaller jobs. This concept is known as flexibility. Assuming that a job has a running time t if simulated in a certain number of processors p, Feitelson and Rudolph [4] propose the following classification regarding the flexibility of jobs (also adopted by [5, 6]):

- ***Rigid jobs:*** They are compiled to be run in a fixed number of processors specified by the user. These applications cannot resize themselves during runtime and is impossible for them to efficiently utilize otherwise idle resources.
- ***Moldable jobs:*** They can be executed on multiple CPU partition sizes, but once the execution starts, these sizes cannot be modified. A task scheduler is in charge of choosing a specific number of processors p from a range of choices given by the user. The decision is made at start time and the number of processors assigned to the job will not change until the end of the execution.
- ***Malleable jobs:*** The number of processors assigned to these jobs can be modified during their execution. A malleable application can grow whenever resources are available and can shrink when resources need to be freed. These changes are requested by the task scheduler depending on the availability of resources and the scheduling decisions.
- ***Evolving jobs:*** The number of processors assigned to these jobs can be modified during their execution, but the changes are requested by the job itself. The task scheduler may accept or deny these adaptation requests depending on the available resources.

Therefore, malleability is able to improve both resource utilization and response time in spite of involving a more complex design of the jobs.

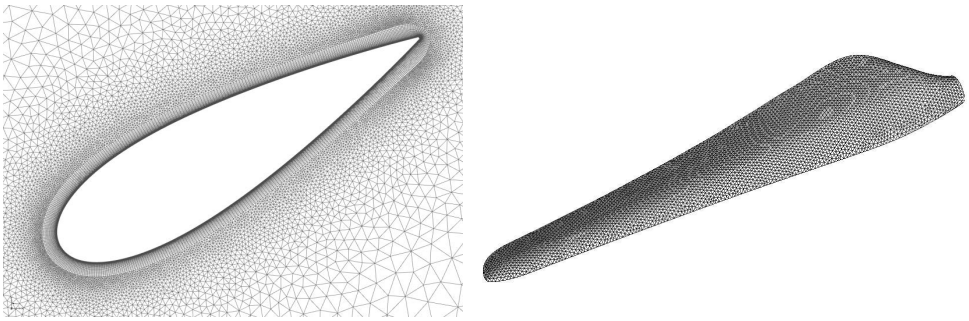
Let us introduce an example belonging to the field of CFD & HT for the sake of a better understanding of the aforementioned concepts. A common optimization problem in wind energy production consists on designing an aerodynamic profile subject to certain structural constraints in order to maximize the power generation in a 3-blade turbine. The design chain involves 3 main steps (see Fig. 3.1):

1. The creation of a two-dimensional airfoil geometry based on a few design variables.
2. Evaluation of the drag and lift coefficients of the airfoil for different angles of attack, checking that the obtained geometries are compliant with structural constraints.
3. Calculation of the output power of a 3-blade wind turbine constructed using the aerodynamic profile obtained in step 1.

Regarding load divisibility, an indivisible task would have to compute all calculations in a single processor and would be independent of any other job (see Fig. 3.2 (a) and Fig. 3.3 (a)). Nevertheless, this design procedure has 3 clearly differentiated steps or modules, which makes more reasonable to consider the task as modularly divisible. The three subtasks may be computed in different processors and are subject to a precedence relation, being necessary to finish subtask 1 before starting subtask 2, and to finish subtask 2 before starting subtask 3. The completion of the original task involves processing all three subtasks or modules (see Figs. 3.2 (b) and 3.3 (b)). Similarly, each of the subtasks could be further subdivided. Unlike steps 1 and 3, step 2 can be computationally demanding since it may contain CFD and structural finite element simulations based on meshed models. Thus, module 2 is an arbitrarily divisible task which can be divided into any number of load fractions with an upper limit equal to the size of the meshes. This is achieved by partitioning the meshes into several subdomains, which are then assigned to different processors (see Figs. 3.2 (c) and 3.3 (c)).

However, strong precedence relations arise when working with meshes. In the example of Fig. 3.4, a mesh composed by 6 cells is depicted and it is assumed that a CFD simulation taking 3 time steps is carried out. If there are enough available processors and each processor is in charge of processing 1 cell, there will be a total amount of 18 tasks or jobs. The 6 tasks belonging to time step t can be solved independently from one another. But the tasks belonging to the same time step t are dependent on those belonging to time step $t-1$, i.e. a cell is dependent on the results obtained for itself and for its adjacent cells in the previous time step. This establishes a strong precedence relation between tasks.

(a)



(b)

(c)

Figure 3.1: Steps for the optimization of the output power of a wind turbine [7]: (a) Design of a two-dimensional aerodynamic profile (b) Evaluation of lift and drag coefficients, check structural constraints (c) Calculation of the output power provided by the 3-blade turbine.

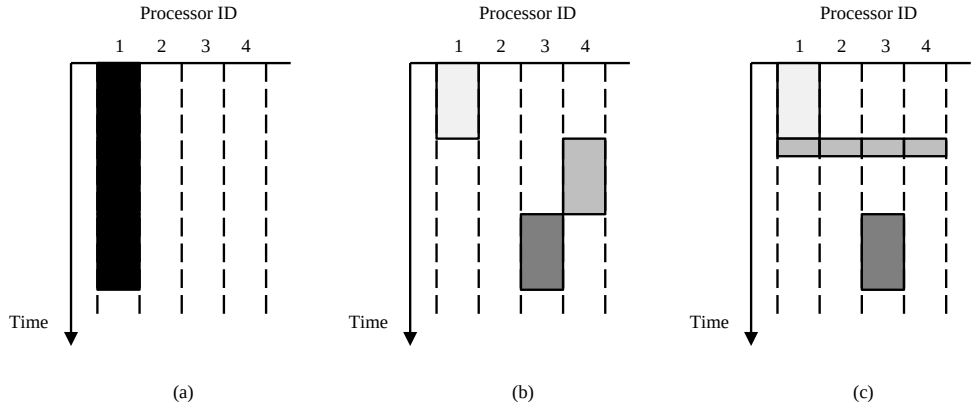


Figure 3.2: Examples of load divisibility: (a) Indivisible load (b) Modularly divisible load (c) Modularly divisible load where the second module is also arbitrarily divisible.

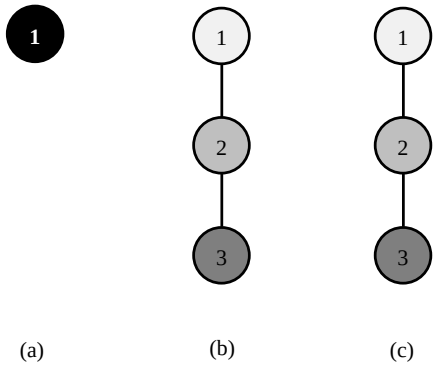
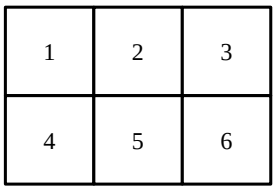


Figure 3.3: Interaction graphs of the tasks represented in Figure 3.2.



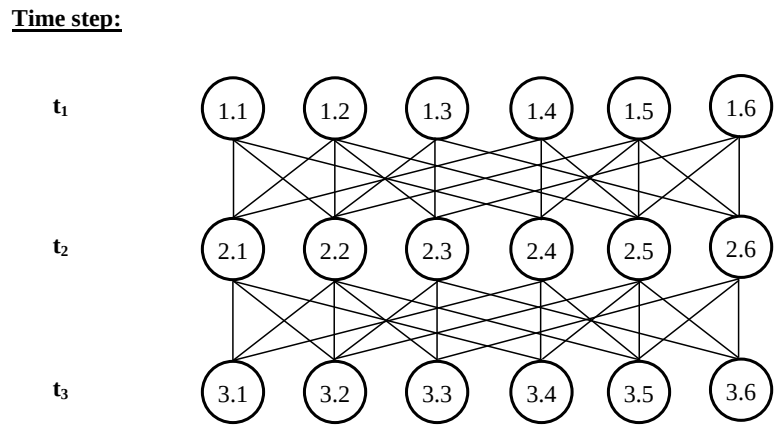


Figure 3.4: Example of the task interaction graph of a mesh with 6 cells simulated in 6 processors during 3 time steps.

Processing a mesh is an arbitrarily divisible job whose flexibility is constrained by the solver. A rigid job would be that which can only be run using a fixed number of processors, 6 for instance. This is comfortable for the designer of the solver in charge of processing the mesh, because it is enough to partition the domain only once and this can be done before beginning any simulation. Nonetheless, the task scheduler of the optimizer would not be able to modify the amount of resources assigned to the task and a degradation of the parallel efficiency could happen when evaluating a batch of individuals. If a moldable job was provided, the task scheduler would be able to decide the amount of processors assigned to each task at the beginning of the evaluation of the batch of individuals. The decision could also be changed at any time before the execution of the task started. This would involve the task to have the ability to call a mesh partitioning module in order to fit the mesh topology to the number of processors assigned dynamically by the task scheduler. A malleable job owns the preferred and most complex degree of flexibility, thanks to the ability of being expanded or shrunk by the task manager in order to adapt to the available resources in run time. This degree of flexibility maximizes the overall parallel efficiency, but also requires a more complex implementation. Finally, an evolving job would be similar to the malleable one with the only difference that the job itself would be able to ask for resources to the task scheduler.

3.2.2 Factors affecting parallel performance

The importance of maintaining a balanced load among the processors in order to achieve high parallel performance in large-scale multiprocessor systems has been already mentioned. The reason is that the total execution time of a set of tasks is determined by the last processor to finish its assigned work. The main factors affecting the parallel performance of applications, i.e. the crucial factors when designing a task schedule, have been identified and are listed hereafter.

Factors related to the hardware:

- ***Hardware heterogeneity:*** The same task has different completion times for different resources when considering a heterogeneous system [2]. Thus, information on the processors' speed is to be taken into account by the scheduler in order to obtain a good balance of the computational load. A potential solution, similarly to that proposed in [8], consists on testing the hardware before the execution of the real set of tasks is started so that the scheduler owns benchmark results of all participating processor nodes indicating their performance.

- ***Network topology:*** A supercomputer is composed by several multiprocessor nodes connected to each other creating a network. Each node's processors have a common memory space called shared-memory, and all the interconnected shared-memory islands give rise to a distributed-memory network. On one hand, each processor is able to access almost instantaneously the shared-memory owned by its node. On the other hand, communication delays are considerably higher in a distributed-memory environment due to the fact that communications are carried out by message passing [3]. Indeed, communication on the network is becoming the bottleneck for the scaling of parallel applications [6]. Two conclusions may be extracted from these considerations. First, tasks using OpenMP, MPI or hybrid models (OpenMP+MPI) have to be carefully placed upon the machines so that hardware affinities are efficiently handled for optimal performance (processor-to-processor affinity). Second, if the data needed by a task are distributed among the local memories of a distributed-memory network, that parallel task will have an affinity for a subset of the processors based on the locality of its memory references (task-to-processor affinity) [9]. Hence, the task scheduler should provide topology aware task placement techniques based on the mentioned processor-to-processor and task-to-processor affinities in order to improve the parallel performance. This feature is even more important in highly parallel applications like mesh-based CFD & HT simulations.

- ***Amount of system's memory:*** As it is mentioned in [9], the amount of memory available in the system can have an indirect impact on the total execution time by allowing the scheduler to modify and expand task-to-processor affinities and to reduce memory conflicts. For example, in a distributed-memory machine, heavily-shared data objects can be replicated in the memories of several processors. This replication expands the number of processors for which a particular task may have an affinity. Disadvantages of this replication are the cost of the additional memory, and perhaps more importantly, the time required to maintain coherence.

- ***Processor partitioning model:*** In the "dedicated master" partitioning, a processor can be dedicated exclusively to scheduling operations. In the "peer partition" approach, the loss of a processor to scheduling is avoided and any processor can run the scheduler when necessary. Obviously, the dedicated master approach seems to be the most suitable model for performing sophisticated scheduling techniques since it exploits the full processing power of the designated processor for this aim [9].

Factors related to the nature of tasks:

- ***Load divisibility:*** A divisible computational load may be split into several tasks, what allows reaching a better load balance but inserts a scheduling overhead caused by the increase in the overall number of tasks. The ability of the scheduler for stopping and restarting a task is also useful as a load division technique. Nonetheless, [10] warns about the overhead of starting a task, which may be due to i) the time to transfer application input/output data to/from each computing resource, and ii) the potential latencies involved when initiating a computation or a communication.

- ***Flexibility of jobs:*** As it has been previously explained, the internal design of arbitrarily divisible parallel jobs can ease or constrain their partitioning into smaller jobs, and this is the concept known as flexibility. The higher the flexibility of the jobs, the easier will be for the task scheduler to balance the overall computational load. Moreover, the use of malleable or evolving jobs allows implementing sophisticated dynamic scheduling algorithms able to adapt the load balance in run time.

- ***Interdependencies between tasks:*** The precedence between tasks is another factor to be taken into account by the scheduler. Although tasks in the batches handled by optimization algorithms are independent and do not need any synchronization or inter-task communications, the use of load division techniques may generate precedence relationships between some of the newly generated tasks.

- ***Heterogeneity in task run times depending on its input variables:*** The tasks contained in each batch handled by an optimization algorithm are of the same type, but their processing time may vary depending on the input variables (the optimization variables). Thus, the task scheduler should be aware of the processing time required by each job. [8] proposes to insert the parallel application into a class having a standardized interface (an enhancement of the FMI interface [11], for instance) that contains information of its performance and parallelizability and with which the task scheduler should be able to interact.

Factors related to the scheduler's configuration:

- ***Limited scheduling time:*** As mentioned in [9], the time required to schedule the tasks can be in direct conflict with the desire to balance the computational

load. For instance, using small task sizes can lead to good load balancing since these small tasks can be used to fill in processor idle times. However, if some of the scheduling operations are performed at run time, the time required to perform the scheduling will be added directly to the execution time of each task. Since this scheduling time is generally independent of the execution time of the parallel tasks, the use of small task sizes can produce very high scheduling overheads, which can negate the advantages of using small tasks for load balancing. Hence, it is useful to limit the scheduling time in such a way that equilibrium between the computational load balance and the scheduling overhead is achieved.

- ***Distributed scheduling:*** The scheduling operation can also be parallelized and distributed among several processors so as to get better task-to-processor maps by assigning greater computational power to the scheduler. However, it must be taken into account that the information required by the scheduler should be available in the memory of the processors in charge of running it, what may involve additional communication overhead [9].

3.2.3 Applications using task schedulers

The need of task scheduling algorithms is shared by several fields in the information technology. Operation of High Performance Computing (HPC) platforms may be one of the best known applications, but other emerging fields like cloud computing and grid computing have shown similar necessities.

There is vast literature analyzing the algorithmic and last advances of Resource and Job Management Systems (RJMS), the software controlling the behavior of HPC platforms. Such system managers take into consideration several features for scheduling [6], like hierarchical resources, topology aware placement, energy consumption efficiency, quality of services, fairness between users or projects, etc. An important metric to evaluate the work of a RJMS on a platform is the observed system utilization. However, studies and logs of production platforms show that HPC systems generally suffer of significant underutilization rates. According to [6], less than 65% of the overall capacity of clusters is utilized throughout the year.

Task scheduling is also a critical problem in cloud computing [2]. A cloud system could have plenty of users that may come from all over the world. Therefore, large-scale task scheduling happens frequently, seeking the cloud providers to reduce the total completion time of the tasks sent by the users.

Grid computing shares computational power and data storage capacity over the

Internet and the goal of grid task schedulers is to achieve high system throughput and to match the applications' needs with the available computing resources [12]. Scheduling on a grid is characterized by three main phases. Phase one is resource discovery, which generates a list of potential resources. Phase two involves gathering information about those resources and choosing the best set to match the applications' requirements. In phase three the task is executed, which includes file staging and cleanup.

Being the characteristics of these three applications (HPC platforms, cloud computing and grid computing) similar to those owned by the process of scheduling the evaluation of individuals from a batch created by an optimization algorithm, literature related to the mentioned fields has been considered a good starting point for knowing the state of the art of scheduling algorithms.

3.2.4 Methodology for developing load balancing algorithms

An overview of the methodology that has been followed for developing and testing load balancing algorithms is explained in this section and can be summarized in four main points:

1. Some assumptions have been made regarding the parallel performance factors considered by the task scheduler in order to simplify the scheduling problem. Complexity will be increased by adding more factors gradually.

2. The load balancing problem has been split in 2 subproblems: tasks' time estimation problem and task scheduling problem.

3. Metrics to evaluate the goodness of the results provided by the load balancing algorithm have been defined.

4. The way of modeling the execution of real tasks in a real High Performance Computing (HPC) system has been decided with the aim of ensuring the reproducibility of experiments as well as reducing their computational cost.

Assumptions on the parallel performance factors

Some assumptions on the parallel performance factors have been made with the aim of simplifying the scheduling problem. As scheduling algorithms that achieve satisfactory results are provided, the maximal complexity of the scheduling problem can be reached by modifying these assumptions gradually.

Assumptions related to the hardware:

- Hardware heterogeneity: The hardware is considered to be homogeneous and scheduling tests are carried out in a single multiprocessor node composed by 32 processors. Thus, it is not necessary to provide any benchmark case for testing the performance of hardware components.
- Network topology: Using a single multiprocessor node implies that all processor-to-processor and task-to-processor affinities are equal. Since all processors have access to a common memory space (shared-memory), communications are quasi-instantaneous between any two of them without the need of message passing.
- Amount of system's memory: The memory of the utilized multiprocessor node is the only system's memory available and may be accessed by any processor. Hence, it is not possible to increase any task-to-processor affinity by adding memory management algorithms.
- Processor partitioning model: The "peer-partition" approach is used. Since a unique node of 32 processors will be available and initial scheduling algorithms are not expected to reach any considerable level of sophistication, it has been thought that sacrificing 1 processor for full-time scheduling is a greater waste of effective computational time than slowing down slightly the completion of some tasks in case a scheduler is run before their computation is started.

Assumptions related to the nature of tasks:

- Load divisibility: Every task is considered to be arbitrarily divisible by the scheduler, which can assign to the task a number of processors ranging between a minimum and a maximum defined by the user. The possibility of stopping and later restarting a task is not allowed, because the restart feature is not that common and/or it may involve a noticeable computational time overhead. Hence, the only way of dividing a task that has been considered is its parallel execution.
- Flexibility of jobs: Every task is considered to be moldable, i.e. the scheduler assigns a number of processors to the task before its execution starts and this number remains constant until the completion of the task.
- Interdependencies between tasks: All tasks belonging to a batch of individuals created by the optimizer are independent from each other, existing no precedence

relations among them. Since it has been assumed that tasks cannot be stopped and restarted, precedence relations will be neither generated in run time.

- Heterogeneity in task run times depending on its input variables: It is possible to have tasks whose processing times vary according to the values of the input variables.

Assumptions related to the scheduler's configuration:

- Limited scheduling time: It is necessary to limit the scheduling time in order to avoid an overall performance degradation caused by excessively long scheduling periods.
- Distributed scheduling: The implementation of sequential task schedulers is foreseen. When satisfactory results are obtained, the possibility of parallelizing those schedulers will be considered.

Splitting the load balancing problem in 2 subproblems

Two steps are clearly differentiated in the resolution of the load balancing problem of a batch of individuals created by an optimizer and solved in a High Performance Computing (HPC) system: the estimation of the tasks' evaluation times and the resolution of a combinatorial problem.

The estimation of the tasks' evaluation times consists on obtaining guess values of the amount of time required by each task if it is simulated in any subset of the available processors. As a result, a map of times is obtained for each task.

The resolution of the combinatorial problem consists on designing a task schedule such that the makespan of the batch of individuals is minimized. The task scheduler uses the time maps calculated for every task in the previous step to estimate the overall simulation time of successive schedules and to try to optimize them by deciding where (in which processors) and when each task is to be processed.

Definition of metrics for evaluating the goodness of a task schedule

The definition of adequate metrics is important in order to evaluate the goodness of a method. In the case of Resource and Job Management Systems (RJMS) used in computing clusters, a typical metric to evaluate the performance of the scheduling algorithm is the observed system utilization. Nevertheless, there is a major difference between the queue of tasks managed by a RJMS and that managed by the task scheduler of an optimizer: the latter has a finite number of similar tasks, whereas the RJMS receives a

continuous flow of heterogeneous tasks coming from different users. Consequently, the makespan of the batch of tasks (i.e. the overall execution time taking place between the start time of the first processed task and the end time of the last task) is found to be a more representative metric than the observed system utilization in order to measure the quality of the obtained schedules for the batches of tasks to be evaluated by the optimizer.

Modeling real tasks in an HPC system

During the development of task schedulers, there are several reasons to avoid testing them by processing real tasks in a production HPC platform: considerable computational load of real tasks, long queues due to tasks sent by other users, variable environmental conditions in the HPC platform affecting negatively the reproducibility of the tests, the fact that specific experiments may need specialized software which is often hard to install on production platforms, etc. Thus, it is advisable to model the tasks to be processed as well as the aspects of the HPC platform to be taken into consideration.

On one hand, and according to [6], tasks can be modeled with up to 3 fidelity degrees depending on the specific features under evaluation:

- ***Sleep applications:*** They consist on using the Unix command "sleep" followed by a number which defines the amount of time in seconds that a processor will stay idle. The main advantage of sleep jobs is that they represent the simplest type of application with a predefined steady duration that can be performed in any number of processors. However, this kind of jobs is not influenced by CPU, bandwidth, or memory stress.

- ***Synthetic applications:*** They are defined based on profiles of real applications and are commonly used with the goal of testing specific parts of the system like CPU, memory, network or I/O. They implicate real computation but do not capture the whole complexity of real applications. Some typical synthetic applications are the following: NAS NPB3.3 benchmarks, Linpack-HPL, pchksum benchmark, Mandelbrot set, etc.

- ***Real applications:*** They hold real-life's complexity and are therefore the most representative way of testing the designed algorithms.

On the other hand, [6] also proposes 3 ways of studying the behavior of an HPC platform:

- ***Emulation of an HPC platform:*** This method consists on utilizing a tool able to execute the actual software of the distributed system in its whole complexity using only a model of the environment. Reproducibility and other external factors are very easily controlled, and it is possible to obtain fast results for the validation of prototype schedulers.
- ***Experimental HPC platform:*** Experimental platforms have been designed and developed to provide better control and reproducibility for real experimentation. This method allows capturing entirely the complex behavior and interaction between the different hardware and software modules and is suitable for validating the observations extracted from the HPC emulator.
- ***Production HPC platform:*** The full complexity of task scheduling is taken into account by means of real-life experimentation without the possibility of manipulating any characteristic of the HPC platform. Conclusions about the quality of the scheduling algorithms are finally extracted.

The ideal experimentation methodology consists on going step by step through all 3 stages: emulator, experimental HPC platform and production HPC platform. Progressively, the control over external factors is lost and the overall computational cost is increased. Once satisfactory results have been obtained, final validation tests can be performed in a production HPC platform. The fidelity degree of the tasks is also expected to be increased gradually.

In the scope of this Doctoral Thesis, no more than two task fidelity degrees have been used: sleep applications and real applications. Regarding the HPC platform, tests have been run in experimental and production systems. The availability of an emulator would have been interesting, but this option was left aside because CTTC had no previous experience with this kind of software.

3.2.5 State of the art of algorithms for solving the combinatorial scheduling problem

Classical combinatorial problems

The study of combinatorial optimization started more than a century ago and gave rise to a set of well-known classical problems: flow shop scheduling, optimization of resource

allocation and leveling, classical knapsack problem, assignment problem, quadratic assignment problem, traveling salesman problem, bin packing problem, etc. Although none of them is directly concerned with scheduling a batch of individuals created by an optimizer, the following three show a certain degree of similarity: the classical knapsack problem, the assignment problem and the bin packing problem.

A set of items, each with a weight and a value, is given in the ***classical knapsack problem***. The objective is to determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. Indeed, the combinatorial task scheduling problem to be solved is quite similar to the ***multidimensional knapsack problem*** and the ***multiple-choice knapsack problem*** (the reader is referred to [13] for further information).

In its most general form, the ***assignment problem*** has a number of agents and a number of tasks and any agent can be assigned to perform any task, incurring some cost that may vary depending on the agent-task assignment. It is required to perform all tasks by assigning exactly one agent to each task and exactly one task to each agent in such a way that the total cost of the assignment is minimized.

In the ***bin packing problem***, objects of different volumes must be packed into a finite number of bins or containers, each of volume V, in a way that minimizes the number of bins used. In the ***two-dimensional bin packing problem*** we are given a set of n rectangular items $j \in J = \{1,\ldots,n\}$, each having width w_j and height h_j, and an unlimited number of finite identical rectangular bins, having width W and height H. The problem is to allocate all the items to the minimum number of bins without overlapping, with their edges parallel to those of the bins. It is assumed that the items have fixed orientation, i.e. they cannot be rotated. The reader is referred to [14, 15] for a review on approximation algorithms proposed for bin packing problems.

However, the heuristic algorithms proposed for solving the above problems are either little robust or the problem for which they were designed is too different from the task scheduler studied in this Doctoral Thesis. Consequently, it was decided to focus the search on more sophisticated algorithms used in the field of High Performance Computing (HPC).

Scheduling policies for RJMS in HPC systems

The Resource and Job Management System (RJMS) is a critical part of any High Performance Computing (HPC) system and is in charge of scheduling the tasks sent by different users. Software development for the RJMS is a major research topic among computers scientists and a large amount of literature is available. The aim of this

section is to collect the most interesting concepts and algorithms of production task schedulers, as well as the newest proposals from research institutions.

Parallel performance factors that should be considered by task schedulers were enumerated in a previous section, together with some assumptions for the particular problem faced in the scope of this Doctoral Thesis. However, many scheduling algorithms in the literature consider only a few of these factors simultaneously. Depending on which factors they take into account, it is possible to group them in categories [9]:

- ***Static vs. Dynamic scheduling***
  If the characteristics of all tasks were known prior to run time, static scheduling could use a complex algorithm to perfectly balance the computational load by appropriately assigning tasks to processors. However, this hypothesis is quite unusual and static algorithms are outperformed by dynamic algorithms, which allow performing the scheduling process at run time when a large amount of useful information about the tasks and their interaction with the underlying architecture becomes available. The only drawback of dynamic schedulers lies on the fact that the overhead associated with the task of scheduling can affect the overall parallel performance.

- ***Centralized vs. Distributed scheduling***
  Centralized techniques store global information at a centralized location and use this information to make more comprehensive scheduling decisions using the computing and storage resources of one or more dedicated processors. A possible drawback is the communication overhead when accessing the shared information and requesting tasks for execution. The alternative consists on having the scheduling task and/or the scheduling information distributed among the processors and their memories, although maintaining a single shared global queue. To prevent more than one processor from executing the same tasks, access to this queue must be limited to one processor at a time using an appropriate synchronization mechanism. A drawback is that the time required to access the queue introduces some run time overhead, but the improved load balancing compared to static scheduling may compensate for this additional delay.

- ***Sequential vs. Overlapped scheduling/execution***
  A system composed by a scheduling algorithm and a set of processors executing the parallel tasks operates in two phases, namely the scheduling phase and the execution phase. In the scheduling phase, tasks are assigned to the processors. During

the execution phase, the working processors execute the tasks that were assigned to them. The scheduling and execution phases may be i) sequential, in which case a schedule of all tasks is planned before executing a single task, ii) interleaved, in which case some processors remain idle while partial task assignments are computed by the scheduling algorithm and being placed on processor queues for execution, and iii) overlapped, which differs from the interleaved configuration in that the overhead of the scheduling effort is completely masked by the execution of previously scheduled tasks.

Regarding the RJMS software, both proprietary and open-source solutions are available. On one hand, and according to [6], the most utilized commercial RJMS systems seem to be Loadleveler, LSF, Moab and PBSPro. On the other hand, the open-source solutions CONDOR, SGE, Torque, Maui, SLURM and OAR own a good reputation. A review of the task scheduling policies implemented in the aforementioned RJMS software together with the last advances in this research field are presented in the following paragraphs. The objective is to obtain a well-defined state of the art, based on which it is possible to design a task scheduler in charge of balancing the parallel evaluation of the individuals created by an optimizer.

Most of the scheduling algorithms used until now in production HPC systems do not involve the use of any complex method. The most widespread solutions are: ***Max-Min*** [12], a common heuristic method whose rationale consists on scheduling the largest tasks at the earliest possible time in order to minimize the overall completion time; ***Min-Min*** [16], in which the task that can be completed the earliest is scheduled each time; ***Min-Max*** [16], which first runs a min-min heuristic for selecting a task and then places that task in the slowest available machine; ***First come first served (Fcfs)*** or ***First In First Out (FIFO)*** [5, 6], in which jobs are processed according to their arrival order; ***Easy-backfilling*** [5, 6], which fills up empty holes in the scheduling tables without modifying the order or the execution of previously submitted jobs; ***Gang Scheduling*** [6], under which multiple jobs may be allocated to the same resources and are alternately suspended/resumed letting only one of them at a time have dedicated use of those resources for a predefined duration; ***Gang scheduling with easy-backfilling*** [5], a hybrid of both constituent methods; ***First come first served-malleable (Fcfs-VM)*** [5], which schedules virtually malleable tasks according to the Fcfs policy; ***Backfilling-Malleable (Backf-VM)*** [5], which schedules virtually malleable tasks according to the backfilling policy; ***Time sharing*** [6], which permits to allocate multiple jobs to the same resources allowing the sharing of computational resources; ***Fairshare*** [6],

which takes into account past executed jobs of each user and gives priority to users that have been less using the cluster; ***Preemption*** [6], which allows suspending one or more "low-priority" jobs to let a "high-priority" job run uninterrupted until it completes; ***Advanced reservation of resources*** [6], which allows reserving certain processors for a certain job to which a precise starting time is assigned; ***Longest Job to Fastest Resource – Shortest Job to Fastest Resource (LJFR-SJFR)*** [16], which assigns the longest task to the fastest resource and the shortest task to the fastest resource alternately; ***Sufferage*** [16], which is based on the idea that better mappings can be generated by assigning a machine to a task that would "suffer" most in terms of expected completion time if that particular machine was not assigned to it; and ***WorkQueue*** [16], which selects a task randomly and assigns it to a machine as soon as it becomes available.

However, research in the field is progressing and has already provided some interesting alternatives to the traditional scheduling methods stated above. A handful of interesting studies is mentioned hereafter.

The ***Self-Adjusting Dynamic Scheduling (SADS)*** class of dynamic scheduling algorithms for distributed memory architectures is introduced in [9], which simultaneously addresses load balancing and memory locality while the duration of each scheduling period is self-adjusted based on the loads in the working processors. Its search engine is based on the branch-and-bound algorithm, it allows both centralized and distributed scheduling strategies, and the scheduling process can be overlapped with the execution of tasks. The scheduler may also be interrupted at any time and is able to provide a valid partial schedule of tasks.

While most load balancing methods assume that each CPU can execute a single parallel process, a different approach that makes load balancing less problem-dependent is used in [17]. It is assumed that each CPU can execute many processes, and load balancing is achieved by moving processes among CPUs. A load balancer has been assigned to each parallel job and is in charge of balancing its corresponding single parallel job independently. It is noted that load balancers may interfere with each other causing conflicts of interests. Hence, the dynamic load balancing problem to be solved is how to map the parallel jobs to processors in such a way that each newly added parallel job is balanced but without affecting negatively the balance of the previously scheduled jobs. The round robin load balancing algorithm for multiple parallel jobs is used for solving the problem.

In [18] genetic algorithms are presented as useful meta-heuristics for obtaining high quality solutions for a broad range of combinatorial optimization problems including the

task scheduling problem. Two hybrid genetic algorithms are proposed for solving the problem of scheduling and mapping a precedence-constrained task graph to processors: the ***Critical Path Genetic Algorithm (CPGA)*** and the ***Task Duplication Genetic Algorithm (TDGA)***. However, the possibility of executing each task in parallel is not considered in the scope of this article.

In [19] ***Work Stealing*** is proposed as a dynamic load balancing method to be used in shared-memory clusters. The algorithm has been designed for simulating batches composed by many short sequential jobs (unlike in the scope of this Doctoral Thesis, where time consuming parallel jobs are considered) and communications are carried out by means of Remote Memory Access (RMA), also called one-sided communications. When a processor runs out of jobs, it accesses the shared memory space and steals a task to the processor with the biggest workload. If none of the tasks can be stolen, the processor exits the simulation of the batch. It must be noted that the number of stolen tasks decreases when the master distributes data on the basis of correct information, whereas it increases if tasks are inappropriately scheduled.

Other interesting publications have also been found and are mentioned next. ***Uniform Multi-Round (UMR)*** is introduced in [10], which is a new multi-round algorithm for scheduling divisible loads and applicable to heterogeneous platforms. ***LAGA*** [2] is a load balance aware genetic algorithm with Min-Min and Max-Min methods to solve the task scheduling problem in cloud computing. An ***Adaptive Genetic Algorithm (AGA)*** is used in [12] to solve the grid scheduling problem and said to outperform the Max-Min algorithm. An approach to static task scheduling based on reinforcement learning and a genetic algorithm is proposed in [20]. Article [16] provides a comparison of heuristics for scheduling independent tasks on heterogeneous distributed environments, including a hybrid ant colony optimization, simulated annealing and genetic algorithms. Finally, greedy scheduling algorithms for massively parallel processing systems are mentioned in [21].

After concluding the review of the literature, some aspects regarding scheduling policies for RJMS systems will be highlighted.

In general, and in agreement to what was observed in [9], little research has been done on employing optimization techniques for dynamic task scheduling. The reason might be the perception that the prohibitive computational cost of these techniques will render them ineffective for dynamic scheduling.

A RJMS system has usually scarce information about the jobs waiting in the queue [5]: job identifier, user who sent the job, job's arrival time to the queue, number of requested processors and an estimation of the required execution time. This time

estimation is usually provided by the user and utilized to abort a job in case it is exceeded. If the queuing algorithm makes use of backfilling policies, load balancing decisions are also based on the mentioned time estimate.

Algorithms able to provide execution time estimates for parallel jobs are being tested, although it is not common to find them in production platforms. The cause is the difficulty of estimating the real duration of tasks. The main shortcomings of user provided time estimates are two, namely i) the completion time of the task when computed in parallel using different numbers of processors is not available, and ii) users usually overestimate times to prevent their jobs from being cancelled by the RJMS. In the particular load balancing problem analyzed in this Doctoral Thesis, there is a single task being run many times with changing input variables. Thus, a considerable amount of information could be extracted and later used for balancing upcoming batches of tasks.

Finally, a crucial scheduling constraint in many RJMS systems is the tasks' arrival order: if a user sent a task at time t_1 and another user sent another task later at time t_2, the start in the execution of the second task cannot delay the start in the execution of the first task. In the particular case of scheduling a batch of tasks created by an optimizer, no priority criteria need to be applied and tasks can be processed in any order. The task scheduler has more freedom in this way and a better load balance may be achieved.

3.2.6 State of the art of time estimation techniques

Approach to the time estimation problem

The use of an accurate time estimation technique is the second requisite for balancing the computational load of a batch of tasks. Otherwise severe errors could be introduced in task time estimates, spoiling the work of even the best scheduling algorithm and leading to serious load imbalance. However, no rigorous study of time estimation techniques was carried out in the scope of this Doctoral Thesis due to lack of time. Some preliminary considerations have been included with the aim of outlining the whole load balancing algorithm, but a thorough development remains as an open issue.

Any method in charge of estimating the completion time of a task when it is run in a certain group of processors needs to carry out two operations: i) store historical data of previous executions of that task, and ii) create a map of the task's completion time based on the stored data by means of an appropriate modeling technique. Optionally, the user could provide a function able to estimate the task's duration. Nonetheless, that function

would not be aware of the way the task's completion time is influenced by the system in which the task is being simulated (due to heterogeneous processors, etc.). Hence, it is concluded that having a time estimation method integrated in the load balancing algorithm is necessary if it is intended to use sophisticated scheduling methods.

In the particular load balancing problem analyzed in this chapter, i.e. a batch of tasks created by an optimizer, a single task is processed several times by changing its input variables and generates a considerable amount of information that can be stored for constructing a good map of the task's completion time estimates. The factors affecting the completion time of a task were previously studied in this chapter and are the following: hardware heterogeneity, network topology and the task's input variables. Consequently, after each execution of the task the load balancing algorithm has to store the input variables of the task, the processors in which the task was executed and the task's completion time. Once a sufficient amount of information regarding the task's execution has been stored, an appropriate modeling technique must be used in order to construct a map of the task's completion time estimates. These modeling techniques may belong to the following categories: deterministic or probabilistic, and global or local.

On one hand, ***deterministic models*** assign a precise prediction of for how long a task will run by means of a single real value. These are the simplest and most comfortable models, but they ignore much of the available information. On the other hand, ***probabilistic models*** are able to characterize the completion time of a task according to the distribution function of historical data, and can provide the probability for each time to happen. These models make use of the whole information available but are more complex to handle, because the schedule turns to be a probabilistic plan instead of a concrete plan [22]. For instance, it may be found that there is an 87% chance for certain processors to be free at a required time. However, this more accurate representation has the potential to lead to better decisions.

Predicting the distribution of a task's run time is also based on the locality of sampling. On one hand, a ***local model*** encompasses only a region of the overall map of times and is constructed based on samples collected in that reduced neighborhood. On the other hand, all available samples are used for building a ***global model***. Local models outperform the accuracy of global ones in their particular region, but are only valid in that reduced space.

Data fitting methodology

The data fitting method used for modeling the task's completion time involves the construction of an approximation or surrogate model (also called response surface)

using data generated from the original truth model [23]. Any surface fitting process consists of four main steps: i) selection of a set of design points, ii) evaluation of the true response quantities at these design points, iii) calculation of the unknown coefficients in the surface fit model using the response data, and iv) diagnosis of the quality of the constructed response surface.

The selection of design points is a crucial decision as it defines the information that the data fitting method will have available. The use of an inadequate sampling method may lead to insufficient accuracy of the obtained response surface, diagnosed in the fourth step. It could even be decided to reject the response surface unless a certain level of accuracy is reached, because the load balancing algorithm would be unable to design any feasible schedule due to excessive time estimation errors.

Several surrogate diagnostic metrics are presented in [23]: i) simple prediction error with respect to the training data, ii) prediction error estimated by cross-validation (iteratively omitting subsets of the training data), and iii) prediction error with respect to user-supplied hold-out or challenge data. All diagnostics are based on differences between the real value at a point x_i and the surrogate model's prediction at the same point. In the simple error metric case, the points x_i are those used to train the model; for cross validation they are points selectively omitted from the build; and for challenge data, they are supplementary points provided by the user.

Several global data fitting methods which could be applicable in the present case are proposed in [23]:

- ***Polynomial regression [24]:*** First-order (linear), second-order (quadratic) and third-order (cubic) polynomial response surfaces computed using linear least squares regression methods.

- ***Kriging interpolation [25]:*** This class of interpolation model has the flexibility to model response data with multiple local extrema. However, this flexibility is obtained at an increase in computational expense and a decrease in ease of use.

- ***Artificial Neural Networks (ANN) [26]:*** This surface fitting method employs a stochastic layered perceptron (SLP) artificial neural network based on the direct training approach of Zimmerman and is designed to have a lower training (fitting) cost than traditional back-propagation neural networks.

- ***Multivariate Adaptive Regression Splines (MARS) [27]:*** This method partitions the parameter space into subregions, and then applies forward and backward regression methods to create a local surface model in each subregion. The result

is that each subregion contains its own basis functions and coefficients, and the subregions are joined together to produce a smooth, C^2-continuous surface model. The generated surface model is not guaranteed to pass through all of the response data values.

- ***Radial Basis Functions (RBF) [28]:*** Radial basis functions are functions whose value typically depends on the distance from a center point, called the centroid. The surrogate model approximation is constructed as the weighted sum of individual radial basis functions.
- ***Moving Least Squares (MLS) [29]:*** Moving Least Squares can be considered a more specialized version of linear regression models. It is a further generalization of weighted least squares where the weighting is "moved" or recalculated for every new point where a prediction is desired. It works well in trust region methods where the surrogate model is constructed in a constrained region over a few points, but it does not seem to be working as well globally.

Although the aforementioned techniques typically use all available sample points to approximate the underlying function anywhere in the domain, piecewise decomposition may be used to build local response surfaces using a few sample points from each neighborhood. It must be noted as well that kriging, MARS and ANN can be used to model data trends that have slope discontinuities and also multiple maxima and minima.

A similar approach to modeling task completion times is done in [22], but with the particularity of being applied to the queuing system of an HPC platform. In this case, the authors propose to use a Hidden Markov Model (HMM) for modeling the statistical distribution of task times.

Before finishing this state of the art study on time estimation techniques, a few recommendations are extracted from [23]. First, surface fitting methods are practical apparently for problems having a small number of design parameters (with the maximum in the range of 30-50 parameters). Second, kriging seems to be the best alternative from the aforementioned data fitting algorithms provided that there are fewer than two or three thousand data points. In case of interpolating among more points, a Radial Basis Function Network or the Moving Least Squares methods are suggested. The reason why using the kriging model is discouraged is that its construction can take a considerable amount of time when the number of data points is very large. Third and last, the use of MARS is generally discouraged when compared to other available methods.

3.3 Load balancing strategies

3.3.1 Overview

The main steps in the execution of a generic optimization algorithm are represented in Fig. 3.5. The one prone to computational load imbalance is the evaluation of a batch of new individuals, and it is at this point where the proposed load balancing strategies are expected to provide a substantial increase in parallel efficiency.

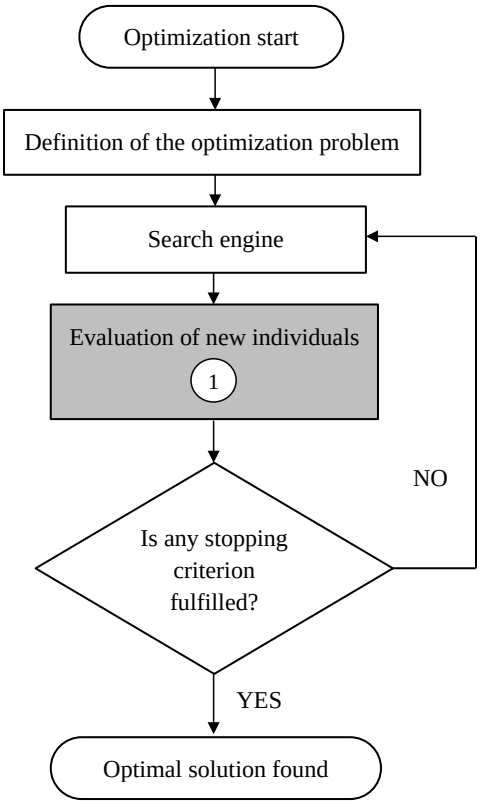


Figure 3.5: Flow chart of a generic optimization algorithm. The evaluation of new individuals is carried out by the task management algorithm (see Figure 3.6).

In the present approach, the load balancing method is considered to be composed by three sub-methods, denominated as: i) task management algorithm, ii) task scheduling

algorithm, and iii) task assignment algorithm. An overview of the three sub-methods, which have been designed to suit the assumptions and requirements stated in section 3.2, is provided hereafter.

Task management algorithm

The task management algorithm (task manager from now on) is the only algorithm interfacing with the optimizer and is represented as a flow chart in Fig. 3.6.

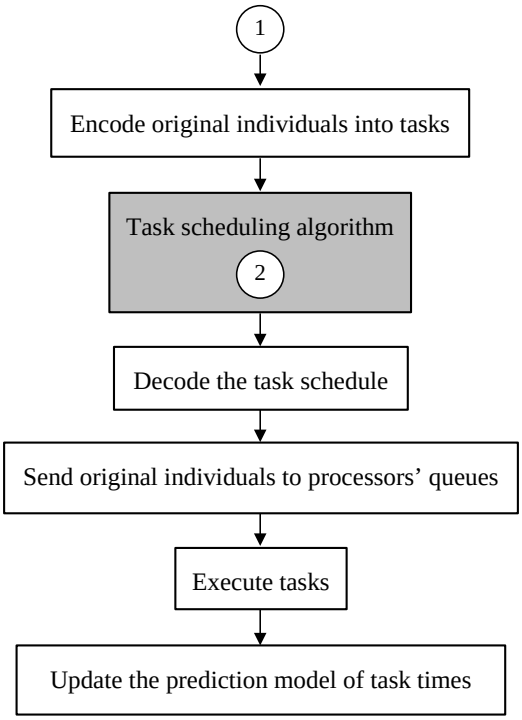


Figure 3.6: Flow chart of a generic task management algorithm. The optimal task schedule is provided by the task scheduling algorithm (see Figure 3.8).

This first sub-method of the load balancing algorithm is called by the optimizer whenever it needs to evaluate a batch of new individuals, and is expected to manage their execution with the aim of minimizing the completion time of the batch. In a first step the individuals to be evaluated (see Fig. 3.7 (a)) are encoded into tasks (see Fig. 3.7 (b)), being a task defined by the following characteristics:

- Task id: It is the task's identifier, necessary to keep the correspondence between the original individuals to be executed and the tasks used for carrying out the load balancing calculations.
- Number of assigned processors: Number of processors in which the task is to be run.
- Ranks of assigned processors: A vector with the ranks (identifiers) of the processors in which the task is to be run.
- Priority: This variable determines the position in the queue of tasks waiting to be mapped to free processors by the task assignment algorithm. See Fig. 3.10 and section 3.3.4 for additional details.
- Run time: Expected completion time of the task.
- Start time: Time at which the execution of the task has been scheduled to start. The reference time $t = 0$ is assigned to the first task of the batch that is executed.
- End time: Time at which the execution of the task is expected to finish.

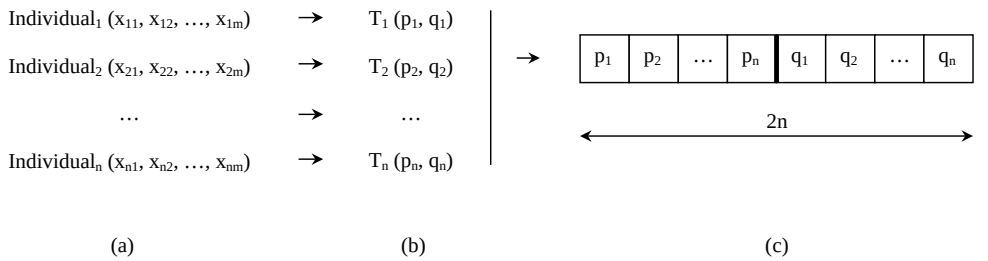


Figure 3.7: Encoding of individuals carried out by the load balancing algorithm: (a) original individuals, (b) batch of tasks, (c) an individual representing the batch of tasks. Variables x_{ij} are the original optimization variables, whereas each task T_i is characterized by the number of processors (p_i) and the priority (q_i) assigned to it.

The only characteristic defined by the task manager is the task id, being the remaining characteristics defined by other sub-methods of the load balancing algorithm.

The next step after creating all the necessary tasks is to schedule them in such a way that they are executed in the right resources and at the right time in order to minimize the completion time of the batch. This step is performed by the task scheduling

algorithm as is explained further on in this section, and the best schedule found is returned to the task manager.

After that, the task manager decodes the received information and has enough data to start executing the real tasks (evaluation of new individuals created by the optimizer), so individuals are sent to wait in the processors' queues until they are called for being computed.

Finally, the task manager gathers information on the processing time of each task in order to feed the task times' prediction model. This step involves a permanent learning process and improvement of task time estimates. When it is finished, the optimizer can continue with the search of optimum until the need of evaluating a new batch of individuals arises, in which case the load balancing algorithm is called again.

Task scheduling algorithm

The task scheduling algorithm (see the flow chart in Fig. 3.8) is called by the task manager and its aim is to decide the resources and the time in which tasks are to be executed in order to minimize the makespan of the batch. This involves the resolution of a combinatorial optimization problem introduced in the following paragraphs but studied with detail in section 3.3.3.

First the batch of n tasks is encoded in such a way that it can be represented by means of a single vector with size $2n$ (see Fig. 3.7 (c)). This is accomplished by representing each task with two variables, namely the number of processors in which the task will be simulated (p_i) and the priority of the task (q_i). These variables have been chosen because they represent unambiguously how the tasks should be scheduled by the task assignment algorithm proposed in section 3.3.4. Thus, the search for the optimal schedule consists on finding the optimal values for the $2n$ variables contained in the vector. The task assignment algorithm is in charge of evaluating the quality (i.e. the makespan) of the schedules successively created by the task scheduling algorithm. Once the optimal task schedule has been found, it is sent to the task manager.

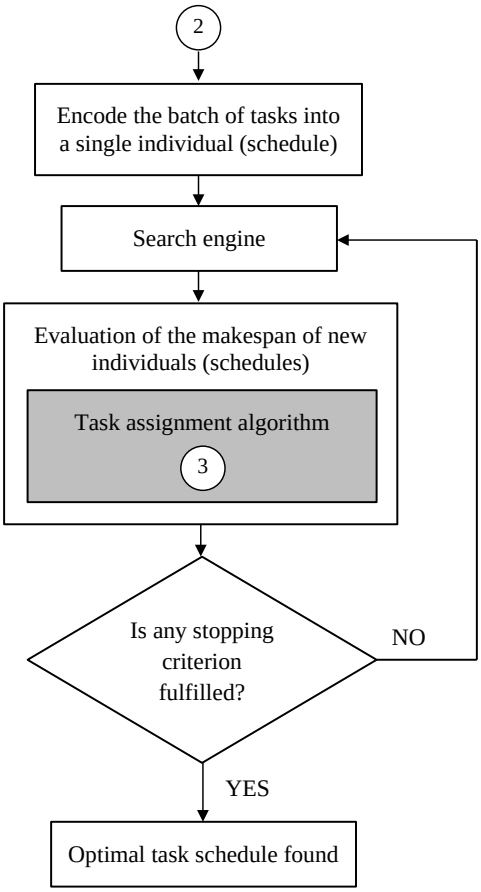


Figure 3.8: Flow chart of a generic task scheduling algorithm. Schedules are successively proposed by the search algorithm and the makespan of each schedule is calculated by the task assignment algorithm (see Figure 3.9).

Task assignment algorithm

The task assignment algorithm is in charge of mapping (assigning) the tasks to the processors based on the schedule it receives from the task scheduling algorithm. This schedule is encoded as it is shown in Fig. 3.7 (c).

According to the flow chart depicted in Fig. 3.9, the first step performed by the algorithm consists on decoding the received task schedule into the task representation shown in Fig. 3.7 (b). Then, the completion time (t_{ij}) of every task (T_i) in the proposed schedule is estimated as explained in section 3.2.6. After that, all the tasks are mapped to the processors based on the number of CPUs and priority which characterize each task, giving as a result the makespan of the batch of tasks. Finally, the makespan of the schedule is sent to the task scheduling algorithm. An illustrative example of the task assignment procedure is provided in Fig. 3.10.

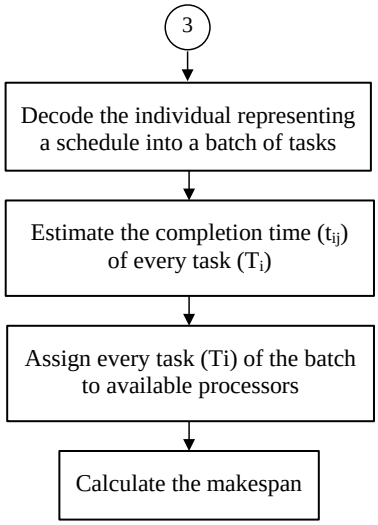


Figure 3.9: Flow chart of a generic task assignment algorithm.

The three sections hereafter are focused on task management, scheduling and assignment algorithms designed to achieve an efficient use of computational resources.

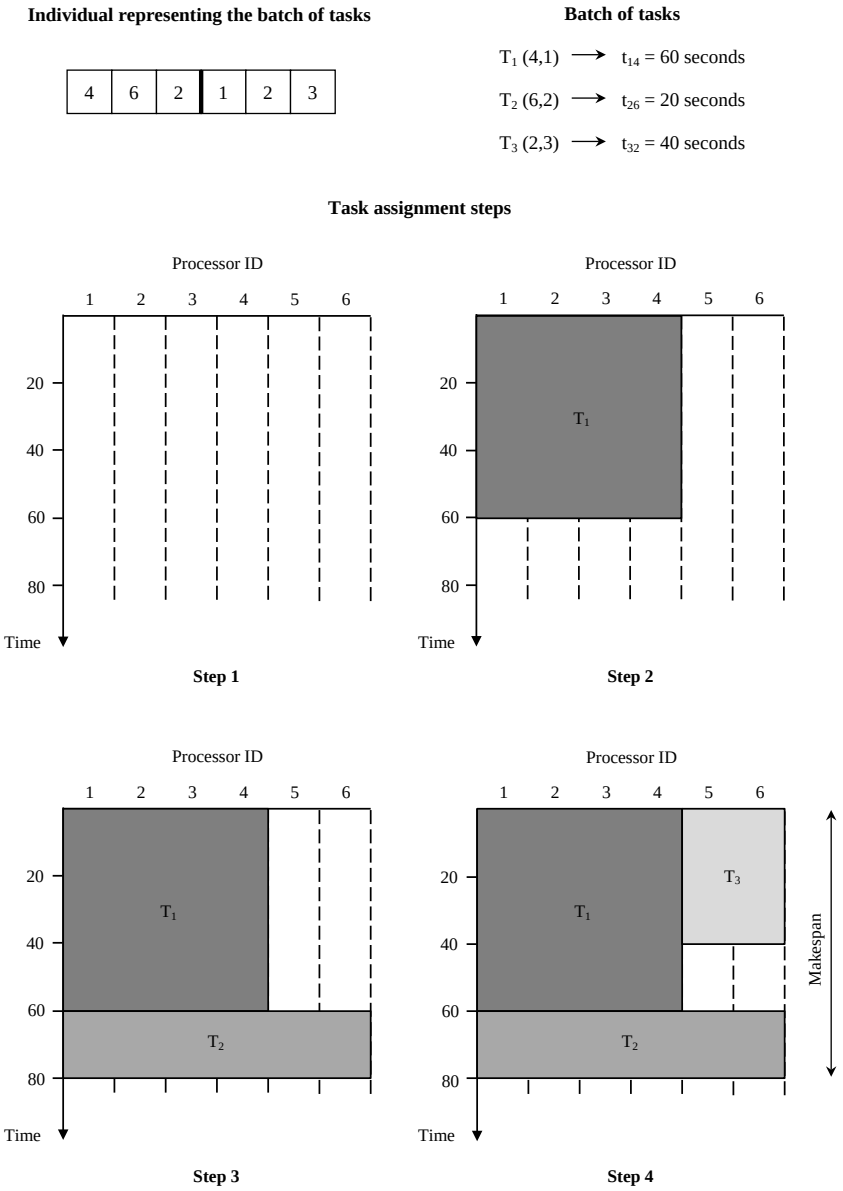


Figure 3.10: Illustrative example of the task assignment procedure by means of which an individual representing a task schedule is decoded, tasks are assigned to processors and the makespan of the batch of tasks is calculated.

3.3.2 Task management algorithms

General aspects

These algorithms are in charge of processing tasks (individuals) in a computationally efficient way every time the optimizer needs to evaluate a batch of individuals. 3 task managers have been implemented: the self-scheduling task manager, the static task manager and the dynamic task manager.

The self-scheduling task manager is the simplest approach and was already presented in section 2.3.8. Every task is executed in the same number of processors, which is defined by the user at the beginning of the optimization. Groups of processors keep on executing tasks from the batch one after the other until it is exhausted. From then on, each group of processors that completes the execution of a task remains idle until all tasks are completed and the optimization algorithm is able to move an iteration forward. At this point, a new population is created by the optimizer and the task manager is called again for its evaluation.

The static and dynamic task managers involve 2 main steps: i) the most suitable processors and time span are found for executing each task (task scheduling), and ii) the batch of tasks is executed. These algorithms are expected to improve the parallel performance of the self-scheduling task manager thanks to a better planning of the tasks' execution and differ from each other in the capacity for adapting to the unexpected behavior of tasks in run time. The static task manager obtains a task schedule at the beginning of the generation's evaluation and cannot be later modified, i.e. the schedule cannot be readjusted in case of unexpected behavior of the tasks. The dynamic task manager obtains the initial schedule in the same way as the static task manager, but tasks are monitored in run time and the task schedule is continuously readjusted until every task in the batch has been completed. Nevertheless, attention must be paid to the following two aspects to make sure that the expected improvement in parallel performance is obtained.

On one hand, the task scheduling operation involves a run time overhead which can negatively affect the overall execution time of the optimizer unless some limitation is introduced. The user can define the maximum allowed time for task scheduling by means of a percentage in both task managers, being the maximum time in generation n calculated as the makespan of generation $n-1$ multiplied by that percentage. Thanks to this strategy, the time for task scheduling is adapted to each generation's execution time.

On the other hand, one of the following two situations may arise: i) a database

with previously simulated task times is already available or the user provided some function able to accurately predict task completion times, or ii) there is insufficient or no information on task times. In the first case, task completion times may be accurately predicted and a good performance of the static and dynamic task schedulers is expected. In the second case, however, a certain amount of individuals needs to be executed in order to be able to obtain accurate time estimations. The early use of either the static or the dynamic task manager is highly discouraged in this case because task schedules based on bad time predictions may have a severe negative impact on the parallel performance of the optimizer.

A sampling method is proposed to generate the database required for time estimation. The method consists on running some initial generations with the self-scheduling task manager until sufficient information has been gathered and then switching to the static or dynamic task manager. It is up to the user to decide the number of sampling generations and the number of processors to be assigned per task in each of those generations. For example, a sampling program defined by the vector $(1,2,3)$ means that 3 generations are to be dedicated to sampling, assigning $2^1 = 2$ processors per task in the first generation, $2^2 = 4$ processors per task in the second generation and finally $2^3 = 8$ processors per task in the third and last generation. From the fourth generation on, the static or dynamic task manager will be activated.

The implemented task management algorithms are described in the following sections.

Coordination of task managers

The main algorithm is referred to as *Algorithm B* and is in charge of the coordination of the 3 available task managers: the self-scheduling task manager (*Algorithm A*, see section 2.3.8), the static task manager (*Algorithm C*) and the dynamic task manager (*Algorithm D*).

Algorithm B *Coordinator of task managers*

---

1. **IF** (it is the first execution of the coordinator of task managers)
   (a) The master processor reads the user-defined vector *samplingDefinition* which contains the sampling program.
2. The master processor sends a message to all other processors communicating the selected task manager.

3. **IF** (processor is the master)
   (a) **IF** (self-scheduling task manager selected)
      i. Send a message to all other processor communicating that no sampling is to be carried out.
      ii. Run the self-scheduling task manager (see *Algorithm A*).
   (b) **ELSE IF** (static task manager selected)
      i. **IF** (there are still sampling generations programmed in *samplingDefinition*)
         A. Send a message to all other processors communicating that a sampling generation is about to start.
         B. Execute the sampling generation (see *Algorithm B.1*).
      ii. **ELSE**
         A. Send a message to all other processors communicating that no sampling is to be carried out.
         B. Run the static task manager (see *Algorithm C*).
   (c) **ELSE IF** (dynamic task manager selected)
      i. **IF** (there are still sampling generations programmed in *samplingDefinition*)
         A. Send a message to all other processors communicating that a sampling generation is about to start.
         B. Execute the sampling generation (see *Algorithm B.1*).
      ii. **ELSE**
         A. Send a message to all other processors communicating that no sampling is to be carried out.
         B. Run the dynamic task manager (see *Algorithm D*).
   (d) **ELSE**
      i. Error: an unknown task manager has been selected and Optimus will be stopped.
4. **ELSE**
   (a) A message containing the selected task manager is received from the master.

(b) A message communicating if a sampling generation is to be executed is received from the master.

(c) **IF** (sampling generation is to be executed)

i. The number of processors to be assigned per task is received from the master.

ii. Run the self-scheduling task manager (see *Algorithm A*).

(d) **ELSE**

i. Depending on the task manager selected by the user, run the self-scheduling task manager (see *Algorithm A*), the static task manager (see *Algorithm C*) or the dynamic task manager (see *Algorithm D*).

Algorithm B.1 *Execute sampling generation*

1. Send a message to all other processors communicating the number of processors to be assigned per task.
2. Run the self-scheduling task manager (see *Algorithm A*).
3. **FOR** (each task in the sampling generation)
   (a) Store in the database (an ASCII text file has been used) the optimization variables, the number of assigned processors and the completion time of the task.
4. Delete the first element of the vector *samplingDefinition*, which corresponds to the sampling generation that has been completed.

Static task manager
The static task manager is described hereafter, being its main algorithm referred to as *Algorithm C*.

Algorithm C *Static task manager*

---

1. The master processor sends a message to all other processors communicating the number of tasks in the batch.

2. All processors create a communicator for the ranks in which the task scheduler is to be executed. To date, only the master (rank = 0) is in charge of running the task scheduler.

3. **IF** (processor is the master)

   (a) Set the maximum allowed time for task scheduling as (*a percentage defined by the user*) · (*the previous generation's makespan*). In case it is the first generation, the maximum time is left unbounded (note that it is later modified in *Algorithm F.2*).

   (b) Run the task scheduling algorithm (see *Algorithm E*) in order to get the optimal schedule and its estimated makespan.

4. Execute the tasks (see *Algorithm C.1*).

Algorithm C.1 *Execute tasks*

---

1. The master processor sends a message to all other processors communicating which processors (ranks) will be involved in executing each task. Note that every processor owns the whole information so that communicators can be created with any MPI function.

2. All processors create a communicator for each task which will be later used by the processors involved in the execution of that task. Thus, as many communicators as tasks in the batch are created. Among the processors in charge of simulating each task, one is designated as the root of the group (usually the processor with the lowest rank).

3. The master sends a sequence of task identifiers to every processor. Each sequence represents the ordered list of tasks to be executed in a certain processor according to the optimal schedule found.

4. For each task to be executed, the master sends to the root of the corresponding group of processors the optimization variables which characterize that task (individual).

5. Every processor evaluates the sequence of tasks assigned to it by executing the task evaluation loop (see *Algorithm C.2*).

6. The results of every task (objective values and simulation time) are sent from the root processors to the master.

7. The master updates the database of task times, adding the optimization variables, the number of assigned processors and the completion time of each task.

8. All MPI communicators are freed.

Algorithm C.2 *Task evaluation loop*

---

1. **FOR** (each task identifier belonging to the sequence)

   (a) The root of the group of processors in charge of executing the task creates a directory in the hard drive in order to store the files needed during the evaluation of the task.

   (b) The group of processors runs the assigned task (evaluates the assigned individual) using the communicator associated to the task for evaluating the objective function(s) in parallel.

   (c) The obtained objective values and the evaluation time are stored in the memory of the root processor.

Dynamic task manager

In comparison to the static task manager, the advantage offered by the dynamic task manager is the adaptation capacity to unexpected task completion times. The behavior of the dynamic algorithm is summarized in the following points:

1. The scheduling and inter-processor coordination are accomplished using 8 vectors stored in shared-memory, which are accessible for any processor by means of one-sided communications.

2. The user decides the number of scheduler runs to be carried out per generation. The first scheduler run always takes place before the evaluation of any individual is started, whereas the remaining runs are uniformly distributed throughout the execution of the batch of tasks. For example, if 15 individuals are to be evaluated and the scheduler is to be run 3 times, it will be run for the second time if at least 5 individuals have been evaluated and for the third time if at least 10 individuals have been evaluated. Note that carrying out one single scheduler run per generation is equivalent to selecting the static task manager.

3. The scheduler is run for the first time by the master and every processor gets a task. Then, the processors assigned to each task create a communicator and start executing the task.

4. When a group of processors completes a task, the root processor checks if the scheduler is to be run. If it is not the case, the root accesses the last schedule and assigns the next task to all processors in the group (each processor may obtain a different task). Then, each processor in the group waits until the other processors necessary for starting the execution of the newly assigned task are free. When this happens, the evaluation of the task is started.

5. In case the scheduler is to be run, the root locks the access to the task schedule to every other processor and it runs the scheduler for mapping the remaining tasks to processors using updated information, i.e. the estimated times at which the already assigned tasks will have been completed in every processor. Once the vectors in shared-memory containing scheduling information have been updated, the root unlocks the access to them. From this point on, the roots of the completed tasks can access the schedule again in order to assign new tasks to idle processors.

6. When the batch of tasks is exhausted, the master processor gathers the objective(s) values, the completion time and the number of utilized processors of each task

and the main optimization process is resumed.

The 8 vectors stored in shared-memory that were mentioned in the first point are listed hereafter, together with an explanation of the information they contain.

- *completedTasks* (vector of Boolean variables): Its size is equal to the number of tasks in the batch and is used for controlling which tasks have been completed and which are still pending. Each position of the vector may hold a 1 (task completed) or a 0 (task pending).
- *pendingTasks* (vector of real variables): It stores scheduling information of all pending tasks, i.e. tasks that have been scheduled but have not been put in the queues of the involved processors, so they could be rescheduled. The size of the vector is calculated as *1 + (number of tasks in the batch) · (total amount of processors + 5)*. The first position of the vector stores the number of pending tasks. From the second position on, each pending task is defined by a block of positions which contains the following information:
  - The identifier of the task (1 position).
  - Number of processors required for executing the task (1 position).
  - Ranks of the processors in charge of executing the task (the number of positions is equal to the number of processors required for executing the task and a rank is stored per position).
  - Estimated completion time of the task (1 position).
  - Identifier of the scheduler run in which the task was scheduled (1 position). It is calculated as *(total number of programmed scheduler runs) - (number of scheduler runs missing when the task was scheduled)*.
  - Task's start time (1 position), i.e. the estimated time at which the task is expected to start its execution.
- *acceptedTasks* (vector of real variables): It stores scheduling information of all accepted tasks, i.e. tasks that have been scheduled and put in the queues of the involved processors but have not been completed. The size of the vector is calculated as *1 + (number of tasks in the batch) · (total amount of processors + 5)*, being information structured in the same way as in the *pendingTasks* vector.

- *unemployedCores* (vector of Boolean variables): Its size is equal to the number of processors and is used for controlling which of them are idle (unemployed), i.e. they do not have any assigned task and will remain in this state unless the scheduler is run again. Each position of the vector may hold a 1 (unemployed processor) or a 0 (busy processor).

- *estimatedTaskTimes* (vector of real variables): Its size is equal to the number of tasks in the batch and is used for storing the estimated completion time of each task. The position of a task in the vector corresponds to its identifier. Every time the scheduler is run, the time estimations of the involved tasks are updated.

- *executionEndtimePrevisionPerCore* (vector of real variables): Its size is twice the number of available processors and is used for storing the estimated time at which each processor will have completed the tasks that has accepted so far. Each processor is referred to in two positions of the vector, e.g. there is information concerning the first processor in the first position of the vector and also in the position (*first position + total number of processors*). The position in the first half contains the estimated time at which all tasks accepted by the processor will have been completed, assuming they are executed with no idle time spans among them. The position in the second half contains the total execution time of the accepted tasks which have not been started yet, assuming they are executed with no idle time spans between them. The reason for this representation is to be able to recalculate the end time of non-started accepted tasks every time a task has been completed by adding the time in the second half to the current time.

- *schedulerInfo* (vector of integer variables): This vector has 4 positions and is used by the master processor (the processor with rank = 0) to share the information regarding the scheduler with all other processors. The content of each position is the following:

  - Position 1: Time estimation method selected by the user and represented by an integer (the methods available in Optimus are explained in section 3.5).
  - Position 2: Variable controlling if the task time prediction model has to be updated or not. At the beginning of the evaluation of a batch of tasks, a task time prediction model is built provided that there is enough information available. This model is kept until the batch of tasks has been completed and the evaluation of the next batch begins. Value 1 means that the task

time prediction model is to be built, whereas value 0 means that the last calculated model may be loaded.

- Position 3: Variable controlling if there is enough information for building a task time prediction model. Value 1 represents there is enough information, whereas value 0 means the available information is insufficient.
- Position 4: Maximum time available for scheduling, calculated based on the last generation's makespan.

- *schedulerRunsMissing* (integer variable): This variable contains the number of scheduler runs that are left before the execution of the batch of tasks finishes in order to reach the number of scheduler calls defined by the user.

Moreover, the explanation of some additional variables has been added so as to ease the understanding of the dynamic task manager (*Algorithm D*) that is later introduced.

- *nRemainingTasks* (integer variable): Number of tasks to be completed in order to finish the execution of the batch of tasks.
- *taskQueuePerCore* (two-dimensional vector of tasks): The size of the first dimension is equal to the number of processors. The second dimension contains a vector of tasks assigned by the scheduler to each processor, thus varying the size of the second dimension among different processors.
- *nextAssignedTask* (vector of real variables): Every processor stores in this vector the next task it has to execute. The size of the vector is equal to (*total amount of processors* + *5*) and the task's information is structured in the same way as in a block of positions belonging to the *pendingTasks* vector.
- *lastTaskRanks* (vector of integer variables): Every processor stores in this vector the ranks of the n cores involved in the execution of the last task carried out by that processor, being the vector's size equal to n.
- *taskWorld* (MPI communicator): Every processor creates a new communicator for each task it has to execute, and it is stored here until the completion of the task.
- *stopSolvingTasks* (Boolean variable): This variable controls if a processor can exit the execution of the batch of tasks (true) or if it has to continue evaluating tasks (false).

- *schedulerLaunchTime* (real value): The current time is stored in this variable before each call to the task scheduling algorithm, being the reference time $t = 0$ taken at the beginning of the execution of the batch of tasks.

Finally, a description of the proposed dynamic task manager is presented.

Algorithm D *Dynamic task manager*

---

1. Preliminary actions (see *Algorithm D.1*).
2. **IF** (processor is the master)
   (a) The information contained in the vector *schedulerInfo* is obtained and stored in shared-memory so that it is available for every processor.
3. The batch of tasks is executed (see *Algorithm D.2*).
4. The following information is sent to the master from the root of each task: objective(s) values, number of involved processors and the task's completion time.
5. **IF** (processor is the master)
   (a) The following information related to each task is stored in the database of task times: optimization variables, number of cores involved in the execution and the completion time.
6. The memory allocated for variables *pendingTasks* and *acceptedTasks* is freed.

Algorithm D.1 *Preliminary actions*

---

1. **IF** (the first generation of the optimization is being evaluated)
   (a) The following 6 variables are created in shared-memory: *schedulerInfo*, *completedTasks*, *unemployedCores*, *estimatedTaskTimes*, *executionEndtimePrevisionPerCore* and *schedulerRunsMissing*. They remain allocated until the main optimization process is finished.
2. A communicator including every processor is created in the variable *taskWorld*.

3. Every processor rank is stored in the vector *lastTaskRanks*.
4. Set every processor to evaluate the next task (*stopSolvingTasks* = *false*).
5. The master reads the total number of scheduler runs to be performed during the batch (defined by the user) and sends this information to every processor, where the information is stored in the variable *schedulerRunsMissing*.
6. The master communicates the number of tasks in the batch (*nTasks*) to every processor, where it is also assigned to the variable representing the number of remaining tasks (*nRemainingTasks*).
7. The following 2 variables are created in shared-memory: *pendingTasks* and *acceptedTasks*. They remain allocated until the execution of the batch of tasks is finished.
8. The master communicates to all other processors the optimization variables of every individual (task) to be evaluated, as well as the bounds of the optimization variables.
9. Vectors *completedTasks*, *pendingTasks*, *acceptedTasks*, *unemployedCores* and *executionEndtimePrevisionPerCore* are initialized to zero values.

Algorithm D.2 *Execute the batch of tasks*

---

1. **WHILE** (*stopSolvingTasks* is false)
   (a) **IF** (processor is the root of *taskWorld*)
      i. Lock access to shared-memory to every other processor.
      ii. Check the activation of the scheduler (see *Algorithm D.3*).
      iii. **IF** (*schedulerActive* is true)
         A. Carry out some actions before running the scheduler (see *Algorithm D.4*).
         B. **IF** (*nRemainingTasks* is not 0)
            Run the task scheduler (see *Algorithm D.5*).
   (b) Wait until every processor sharing the communicator *taskWorld* reaches this point in the code (equivalent to MPI_Barrier(*taskWorld*)).

(c) The root of *taskWorld* communicates to all other processors sharing that communicator the value of *nRemainingTasks*.

(d) Get the next task to be executed (see *Algorithm D.6*).

(e) **IF** (there is a task contained in the vector *nextAssignedTask*)

i. Execute the next task (see *Algorithm D.7*).

(f) **ELSE**

i. Check the stopping criterion (see *Algorithm D.8*).

Algorithm D.3 *Check the activation of the scheduler*

1. Read the variables *schedulerRunsMissing* and *completedTasks* in shared-memory.
2. A variable named *border* is calculated to decide if the scheduler should be called: *border* = [*nTasks* · *schedulerRunsMissing* / *total number of scheduler runs*].
3. **IF** (*schedulerRunsMissing* > 0 **AND** *nRemainingTasks* ≤ *border*)
   (a) The scheduler is to be run (*schedulerActive* = true).
   (b) The value of *schedulerRunsMissing* is reduced by 1 unit and is updated in the shared-memory.

Algorithm D.4 *Actions before running the scheduler*

1. Read the vector *unemployedCores* from shared-memory.
2. Send the unemployed processors' ranks list to every unemployed processor.
3. Clear the vector *unemployedCores* in the shared-memory.
4. **IF** (*nRemainingTasks* is not 0)
   (a) Save the current time in the variable *currentTime*.
   (b) Read the vector *executionEndtimePrevisionPerCore* from shared-memory.
   (c) **FOR** (every rank contained in *lastTaskRanks*)

i. *executionEndtimePrevisionPerCore[rank] = currentTime + executionEndtimePrevisionPerCore[rank + total number of processors]*.

Algorithm D.5 *Run the task scheduler*

1. **IF** (first time that the scheduler is run in this batch of tasks)
   (a) The maximum allowed scheduling time is calculated based on the last generation's makespan and a percentage introduced by the user: *maxSchedulingTime = userDefinedPercentage · lastGenTime*. The result is saved in the 4th position of the vector *schedulingInfo*, which is stored in shared-memory.
2. Save the current time in the variable *schedulerLaunchTime*.
3. Read the vector *schedulerInfo* from the shared-memory.
4. Calculate a balanced schedule (see *Algorithm E*). As a result, a vector of tasks and an estimated makespan for the batch of tasks are obtained.
5. Update the *taskQueuePerCore* variable, writing the tasks assigned by the scheduler to every processor.
6. Update positions 2 and 3 of the *schedulerInfo* vector, which store respectively the variable controlling if the task time prediction model has to be updated and the variable controlling if there is enough information available for building a task time prediction model. The updated vector is stored in shared-memory.
7. Copy the scheduled tasks to the *pendingTasks* vector ascendingly according to their start time, and update the vector in the shared-memory.
8. Update the vector *estimatedTaskTimes* in the shared-memory with the new time estimations provided by the scheduler.

Algorithm D.6 *Get the next task*

1. **IF** (processor is root of *taskWorld*)
   (a) Read the vectors *pendingTasks* and *acceptedTasks* from shared-memory.

(b) Assign tasks from the vector *acceptedTasks* to processors. For this aim, call to *Algorithm D.9* giving *acceptedTasks* as input variable.

(c) Assign tasks from the vector *pendingTasks* to processors. For this aim, call to *Algorithm D.9* giving *pendingTasks* as input variable.

(d) Move to the vector *acceptedTasks* the tasks contained in *pendingTasks* which have been assigned to processors (see *Algorithm D.10*).

(e) Communicate to all processors in *lastTaskRanks* the next task they have to execute. Then, store in *nextAssignedTask* the next task to be executed in the current processor.

(f) **FOR** (every processor contained in *lastTaskRanks*)

i. If no task has been assigned to the processor, store value 1 in the position of the vector *unemployedCores* (in the shared-memory) corresponding to that processor.

(g) **IF** (there are no more tasks for being scheduled)

i. Avoid the execution of additional schedulers in the current batch of tasks by setting *schedulerRunsMissing* = 0 in the shared-memory and sending it to every unemployed processor.

(h) **FOR** (every unemployed processor)

i. Save the current time in the position corresponding to the processor in the first half of the vector *executionEndtimePrevisionPerCore* (stored in shared-memory).

(i) Update vectors *acceptedTasks* and *pendingTasks* in shared-memory.

2. **ELSE**

(a) Receive from the root of *taskWorld* the next task to be executed. In case no task has been assigned to the processor, receive the updated value of *schedulerRunsMissing*.

Algorithm D.7 *Execute next task*

1. Read in *nextAssignedTask* the details of the next task to be executed.
2. Clear *lastTaskRanks* and save the ranks of the processors involved in the execution of the next task.
3. **IF** (processor is the one which locked the access to shared-memory)
   (a) Unlock the access to shared-memory.
4. Free the *taskWorld* communicator and create a new one for the ranks contained in the vector *lastTaskRanks*.
5. **IF** (processor is the root of *taskWorld*)
   (a) **FOR** (every rank contained in *lastTaskRanks*)
      i. *executionEndtimePrevisionPerCore[rank]* = current time + *executionEndtimePrevisionPerCore[rank + total number of processors]*.
      ii. *executionEndtimePrevisionPerCore[rank + total number of processors]* = previously stored time - *estimatedTaskTimes[next task's identifier]*.
   (b) Delete the information concerning the task in *nextAssignedTask* from *acceptedTasks*, keeping the remaining tasks in the same order.
   (c) Create a directory in which the task can write its execution files.
6. Execute the task (evaluate the objective function(s) of the individual). Store the objective value(s) and the task's completion time in the root of *taskWorld*.
7. Clear the vector *nextAssignedTask*.

Algorithm D.8 *Check the stopping criterion*

1. **IF** (processor is the one which locked the access to shared-memory)
   (a) Unlock the access to shared-memory.
2. Free the communicator *taskWorld*.

3. **IF** (*schedulerRunsMissing* > 0)
   Before the task scheduler is called again, all unemployed processors create a communicator (*taskWorld*) as if they had simulated a task together. They received a list of unemployed processors for this aim in a previous instruction.
   (a) Copy the list of ranks of unemployed processors into *lastTaskRanks*.
   (b) All ranks contained in *lastTaskRanks* create a communicator in *taskWorld*.
4. **ELSE**
   (a) When a processor reaches this point, it is impossible that it gets assigned any other task of the batch. Thus, the variable *stopSolvingTasks* is set to true so that the processor can exit the generation's evaluation.

Algorithm D.9 *Assign tasks to processors*

---

1. An input vector is received with the tasks that are to be assigned to processors (*taskVector*).
2. **FOR** (every task in *taskVector*)
   (a) **FOR** (every processor involved in the task)
      i. **IF** (there is no task assigned to that processor)
         A. Assign the current task in *taskVector* to the processor.
      ii. **ELSE**
         A. If the task proposed from *taskVector* has a smaller start time than the task already assigned to the processor, replace the existing task by the new task. Note that if the already existing task was assigned before the last run of the scheduler, it will not be replaced.

Algorithm D.10 *Move pending tasks to accepted tasks*

---

1. **FOR** (every task in the vector *pendingTasks*)
   (a) **IF** (the task was assigned to processors)

i. Delete the task from *pendingTasks* and write it after the last existing task in *acceptedTasks*.

ii. Store the task as completed in the vector *completedTasks*. Otherwise, the task may be scheduled again.

iii. Update in the shared-memory the execution end time previsions of the processors to which the new task was assigned.
FOR (every processor to which the new task was assigned)

A. *executionEndtimePrevisionPerCore[rank]* = previously stored time + *estimatedTaskTimes[new task's identifier]*.

B. *executionEndtimePrevisionPerCore[rank + total number of processors]* = previously stored time + *estimatedTaskTimes[new task's identifier]*.

3.3.3 Task scheduling algorithm

This algorithm is in charge of solving the task scheduling problem, which is defined as follows:

Definition 3.1 Task scheduling problem: *Given a set of n independent tasks* $\boldsymbol{T} = \{T_1, T_2, ..., T_n\}$ *and being* t_{ij} *the time required by task* T_i *to be run in j processors, where* $1 \leq i \leq n$ *and* $j_{min} \leq j \leq j_{max}$*, find the task schedule that minimizes the makespan of* $\boldsymbol{T}$ *if a total number of p processors is available.* j_{min} *and* j_{max}*, bounded as* $1 \leq j_{min} \leq p$, $1 \leq j_{max} \leq p$ *and* $j_{min} \leq j_{max}$*, represent respectively the minimum and maximum number of processors that can be assigned to a task and are user defined.*

Based on the state of the art study carried out in section 3.2, it has been decided to use a genetic algorithm in order to solve the task scheduling combinatorial problem. The main reason is that results provided by optimization algorithms outperform those achieved by previously used heuristic methods. Other stochastic methods are also suitable for this kind of problems, but the genetic algorithm has been selected because it is already implemented in the Optimus library.

Encoding

The individuals handled by the task scheduling algorithm are encoded as shown in Figure 3.7 (c). Thus, if the schedule of a batch of n tasks is being optimized, each individual is represented by $2n$ optimization variables. Each one of the first n variables (let us say they belong to half A) represents the number of processors assigned to each task, whereas the remaining n variables (let us say they belong to half B) represent the priority of each task to be assigned by the task assignment algorithm.

The size of a combinatorial optimization problem depends on the number of optimization variables and on the number of different values each variable may take. Taking this aspect into account has led to represent the variables in half A (the number of processors assigned to each task) as powers of 2, i.e. 2^x processors are assigned to each task, being x the value for each task contained in half A of the individual. If the user defines the bounds for x to be $[0,5]$, the scheduler may decide to run each task in 1,2,4,8,16 or 32 processors. Variables in half B (the priority of each task) are always natural numbers bounded in $[1,n]$, having value 1 the task that will be first assigned and value n the task to be assigned in the end.

Enhancement of Optimus for combinatorial optimization

The optimization library Optimus implemented in Chapter 2 was only suited for continuous optimization. Therefore, additional features have been included for solving discrete optimization problems (particularly the present scheduling problem) and are listed hereafter.

- The possibility of choosing the nature (real, integer or boolean) of each optimization variable independently has been included, giving rise to the possibility of solving mixed integer optimization problems.
- New crossover operators
  - Order crossover
    This permutation operator can be executed in 5 steps (see example in Fig. 3.11): i) select randomly two cutting points, ii) copy the substring from parent 1 that falls between the two cutting points to the beginning of offspring 1, iii) copy the remaining genes (avoiding duplication) from parent 2 beginning at the position following the second cutting point and respecting their order, coming back to the beginning of the chromosome when the end of the string is reached, iv) copy the substring from parent 2 that falls between the two

cutting points to the beginning of offspring 2, and v) copy the remaining genes (avoiding duplication) from parent 1 beginning at the position following the second cutting point and respecting their order, coming back to the beginning of the chromosome when the end of the string is reached.

- Task crossover
  The encoding of a task schedule was already explained, being the number of processors assigned to each task contained in half A and the priority of each task contained in half B (see Fig. 3.7 (c)). Hence, each task is scheduled according to the pair (p_i,q_i). The task crossover is carried out by applying the order crossover permutation operator to half B, which causes the reordering of the n pairs (p_i,q_i), i.e. a different pair (p_i,q_i) is assigned to each task (see example in Fig. 3.12). Note that the number of processors p_i associated to priority q_i is kept.

- New mutation operators

  - Swap mutation
    A swap consists on interchanging the values of 2 uniformly selected positions in the chromosome, and the user decides how many swaps shall be carried out. This is a common permutation operator.

  - Task mutation
    This operator has been specifically designed for mutating task schedules and applies a deterministic-uniform mutation (see section 2.3.4) on half A of the chromosome and a swap mutation on half B, being both halves independently manipulated. Regarding the mutation on half A, it changes k values in a range of size $2 \cdot epsilon$ and the obtained new real values are rounded to the closest integer, being both k and *epsilon* user defined parameters. Regarding the mutation of half B, the number of swaps to be carried out is decided by the user. The application of the task mutation operator depends on a single probability parameter, so either both or none of the constituent mutation operators are applied.

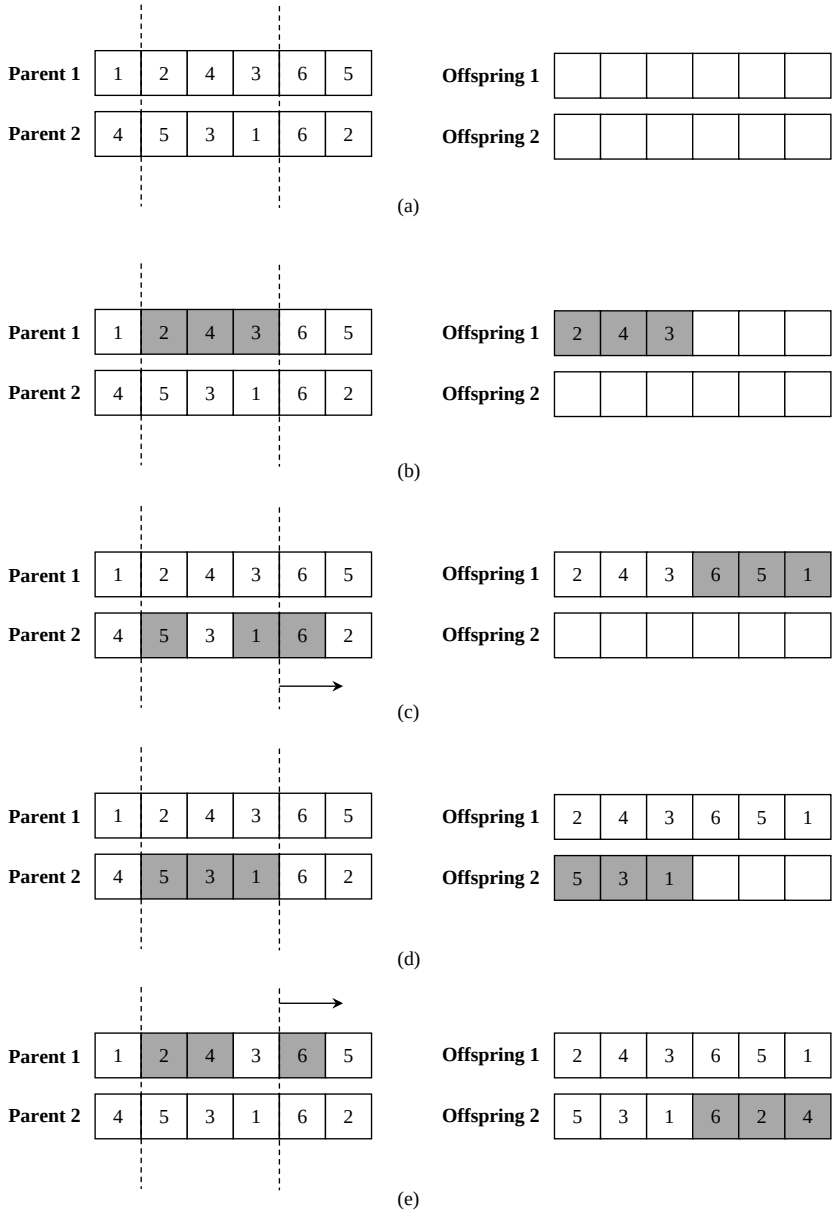


Figure 3.11: Example of the order crossover permutation operator, which recombines 2 parent strings based on 2 randomly chosen cutting points.

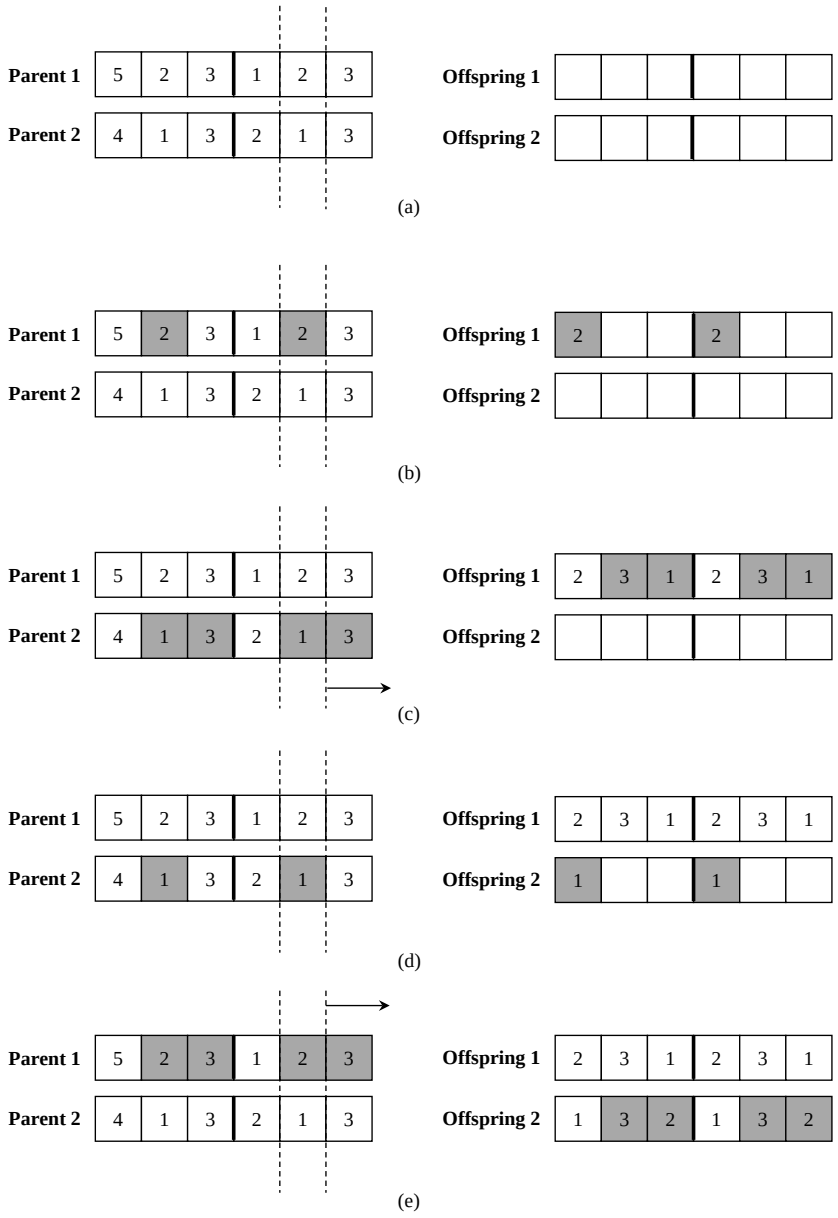


Figure 3.12: Example of the task crossover permutation operator, which recombines 2 parent strings representing task schedules (see Figure 3.7 (c)) based on 2 randomly chosen cutting points.

No exhaustive study has been carried out in order to find the optimal values for the genetic algorithm's parameters, but the following values have led to acceptable results when scheduling batches composed of 20-60 tasks in 32 processors:

- Population: 40 individuals.
- Available processors per task: between 1 and 32.
- Selection algorithm: ordered sequential selection.
- Replacement algorithm: plus replacement.
- Number of offspring: 60% of the population.
- Continuation criterion: stop if the best individual of the population is not improved in 400 generations.
- Probability of crossover: 100%
- Probability of mutation: 100%
- k (number of genes to mutate in half A): 1
- *epsilon* (defines the range of mutations in half A): 5
- Number of swaps (in half B): 1

Description of the algorithm
A description of the proposed task scheduling algorithm is introduced hereafter.

Algorithm E *Task scheduling algorithm*

---

1. Encode the batch of tasks into a single individual (schedule).
2. Run the search engine in charge of minimizing the makespan of the batch of tasks. **WHILE** (none of the stopping criteria is fulfilled)
   (a) Use the search engine to create a new population of potential schedules.
   (b) Calculate the makespan of all potential schedules by means of the task assignment algorithm (see *Algorithm F*).

3. The optimal task schedule has been found.

Observations

It is proved in section 3.4 that the aforementioned genetic algorithm for task scheduling is able to balance computational load satisfactorily. Nevertheless, several shortfalls have been detected and are listed hereafter in order to lead upcoming research:

- Enabling the use of self-adaptive mutation with discrete chromosomes (currently it is only suited for real-valued chromosomes in Optimus) could improve the performance of the genetic algorithm used for task scheduling.
- A thorough study on discrete crossover and mutation operators is needed in order to select the best performing algorithms.
- The size of the scheduling combinatorial problem increases dramatically in case there is a considerable amount of tasks in the batch. Keeping in mind that the risk of load imbalance arises at the end of the batch, a good strategy could be to balance randomly created sub-batches of tasks. These sub-batches would be successively created until the execution of all tasks finished, and the computational cost of optimizing each sub-batch is considerably lower than the cost involved by the whole batch.
- The size of the population should be in agreement with the size of the combinatorial problem to be solved. Thus, some study in this direction is required.
- Since the gradient-based local search method currently available in Optimus (Trilinos/Moocho) is not suited for discrete optimization, the addition of some other discrete local search method could improve the performance of the task scheduling algorithm.

3.3.4 Task assignment algorithm

The task assignment algorithm receives a schedule from the task scheduling algorithm (see Fig. 3.7 (c)) and is in charge of determining the most suitable processors for executing each task, as well as of establishing a precedence order between the tasks to be run in each processor. The tasks are mapped on a one-by-one basis according to their priorities seeking to assign them the earliest possible start time. A graphical example of the task assignment procedure is shown in Fig. 3.10.

The implemented task assignment algorithm is described hereafter. The main algorithm is referred to as *Algorithm F*, and at certain points other sub-algorithms are called. But let us first introduce 2 variables in which the task-to-processors map proposed by the assignment algorithm is stored:

- *batchOfTasks*: It is a vector that includes all tasks in the batch, being each task defined by the characteristics mentioned in section 3.3.1: task id, number of assigned processors, ranks of assigned processors, priority, run time, start time and end time. The notation *batchOfTasks[taskID]* is used for referring to a task with a certain id contained in the batch, being the index *taskID* in the range $[1, n]$.
- *taskQueuePerCore*: It is a matrix of tasks that stores the queue of each available processor. The notation *taskQueuePerCore[rank][i]* refers to the i-th task waiting in the queue belonging to the processor with the identifier *rank*. This matrix contains the information used by the task manager for executing the tasks in the batch.

Algorithm F *Task assignment algorithm*

---

1. The task schedule proposed by the task scheduling algorithm (see Fig. 3.7 (c)) is decoded into a vector of tasks (*batchOfTasks*).
2. The completion time of each task is estimated (see *Algorithm F.1*).
3. The tasks contained in *batchOfTasks* are ordered ascendingly according to their priority.
4. If the dynamic task manager is being used, fictitious tasks are created (see *Algorithm F.5*).
5. **FOR** (each task T_i contained in *batchOfTasks*)
   (a) Call *Algorithm F.6* giving as an input the task T_i in order to map that task to the processors.
6. At this point, every task in *batchOfTasks* has a defined start time, end time and a list of ranks of the processors assigned for its execution. But the assignment procedure is finished when every task is put in the queue of its corresponding processors in *taskQueuePerCore*:

FOR (each task T_i contained in *batchOfTasks*)

(a) **FOR** (each rank assigned to task T_i)

i. Add the task T_i to the queue *taskQueuePerCore[rank]*.

ii. Order the tasks in the queue *taskQueuePerCore[rank]* ascendingly according to the start time of the tasks.

7. The makespan of the task schedule is calculated by comparing with each other the end times of the last task belonging to the queue of each processor and taking the greatest of them, i.e. the makespan is defined by the end time of the last task being processed.

Algorithm F.1 *Task time estimation manager*

1. **IF** (the time estimation method is a function provided by the user)

   (a) Estimate the completion time of all tasks.

   (b) Check the scheduler's run time limitation (see *Algorithm F.2*).

2. **ELSE**

   (a) Check if there is enough information available in the task times' database in order to build a task time prediction model using the generic method selected by the user (for further details on generic time estimation methods implemented in Optimus, the reader is referred to section 3.5).

   (b) **IF** (enough information available in the database)

      i. Estimate the completion time of every task (see *Algorithm F.3*).

   (c) **ELSE**

      i. It is not possible to provide any task time estimation. The optimization is stopped and the user is requested to modify the parallelization parameters.

Algorithm F.2 *Check the scheduler's time limitation*

1. If the first generation is being evaluated, the task scheduler does not have any run time limitation, as it is calculated based on the previous generation's makespan. In this case, the time limitation may be introduced as follows:
   IF (the first generation is being evaluated)
   (a) The completion time of all tasks in the batch is estimated assuming that each of them is executed using every processor.
   (b) The generation's makespan is estimated assuming that all tasks are executed one after the other.
   (c) The run time limitation for the task scheduler is calculated by applying the percentage defined by the user to the generation's makespan estimated in the previous step.

Algorithm F.3 *Estimate task times with a generic method*

1. Check position 2 of the vector *schedulerInfo* stored in shared-memory to know if a task time prediction model is to be built.
2. **IF** (task time prediction model is to be built)
   (a) Build a new task time prediction model using the generic method selected by the user and store it in the hard drive so that it can be loaded by any processor.
3. Load the task time prediction model stored in the hard drive.
4. Estimate the completion time of every task using the loaded task time prediction model.
5. Check the scheduler's time limitation (see *Algorithm F.2*).
6. Check the existence of negative task times (see *Algorithm F.4*).

Algorithm F.4 *Check the existence of negative task times*

1. Depending on the quality of the task time model, it might happen that the time estimates of some tasks get a negative value. In order to avoid the detrimental effect of such estimates in the task assignment algorithm, this algorithm is an alternative for turning negative times into positive times.
2. **IF** (at least one task got a positive completion time estimate)
   (a) Calculate the average completion time of the tasks that got a positive time estimate (t_{avg}) assuming they are executed in a single processor.
   (b) Take each task which got a negative completion time estimate and replace its time estimate by the value obtained when dividing t_{avg} by the number of processors assigned to the task.
3. **ELSE IF** (all task completion time estimates are negative)
   (a) Take the task with the smallest time estimate (the most negative estimate) and calculate a constant K such that if it is added to the task's time estimate, the value 1.0 is obtained.
   (b) Add the same constant K to the completion time estimate of every task, so that all of them obtain a positive value.

Algorithm F.5 *Create fictitious tasks*

1. A fictitious task is created per processor, representing the remaining time for finishing the task being executed in each of them.
   FOR (every processor)
   (a) Create a fictitious task with the following characteristics, being each processor represented by its rank:
   Task id = -1 (this value is assigned to every fictitious task)
   Number of assigned processors = 1
   Ranks of assigned processors = *rank*
   Run time = *executionEndtimePrevisionPerCore[rank] - schedulerLaunchTime*
   Start time = *schedulerLaunchTime*
   End time = start time + run time

(b) Add the created fictitious task to the vector *taskQueuePerCore[rank]*.

Algorithm F.6 *Map task to processors*

1. The input received by this algorithm is a task T_i, which is expected to be mapped to the processors at the earliest possible start time.

2. A gap, represented as $gap(t_{start}, t_{end})$, is defined as an idle time span beginning at a start time and finishing at an end time. In order to map the task T_i to processors, the suitable gaps in each processor (i.e. the gaps involving a sufficient time span so that task T_i can be completed) are identified and stored in a matrix of gaps (let us call it *matrixOfGaps*), where each row contains the gaps of a processor ordered ascendingly according to their start time t_{start}. The following loop obtains one by one each row of *matrixOfGaps*:
FOR (each processor)

   (a) Call *Algorithm F.7* providing 2 inputs: the task T_i and the rank of the processor.

3. Next, an empty vector of gaps (*vectorOfGaps*) is created and every gap found for every processor is copied there. Then all gaps are ascendingly ordered according to their start time t_{start}.

4. **FOR** (each gap_i contained in *vectorOfGaps*)

   (a) Let us represent the start time of $gap_i(t_{start}, t_{end})$ as $gap_i.t_{start}$ and its end time as $gap_i.t_{end}$. Being k the number of processors required by task T_i, search for $k-1$ additional gaps (generically represented as gap_{add}, where index add is greater than index i which fulfill that $gap_{add}.t_{start} \leq gap_i.t_{start}$ and $gap_{add}.t_{end} \geq gap_i.t_{end}$).

   (b) **IF** ($k-1$ additional gaps are found)

      i. Task T_i can be executed in the processors containing those gaps, so the ranks of the processors are stored in the vector *processorsWithSuitable-Gaps*.

ii. The following information is assigned to task T_i contained in *batchOfTasks*:
$T_i.t_{start} = gap_i.t_{start}$
$T_i.t_{end} = T_i.t_{start} + T_i.runtime$
$T_i.ranksOfAssignedProcessors = processorsWithSuitableGaps$

The loop always finds a suitable set of gaps, because each processor can always accept a new task at the end of its queue.

Algorithm F.7 *Search for gaps in a given processor*

1. The inputs received by this algorithm are 2: a task T_i and the rank of the processor in which gaps are to be searched.
2. An empty vector of gaps (*gapsInRank*) is created, which is in charge of storing the gaps that will be found in subsequent steps.
3. **FOR** (each task T_k contained in *taskQueuePerCore[rank]*, excepting the last task T_{last})
   (a) **IF** ($T_{k+1}.t_{start} - T_k.t_{end} \geq T_i.runtime$)
      i. Create $gap(T_k.t_{end}, T_{k+1}.t_{start})$ and add it to *gapsInRank*.
4. Create $gap(T_{last}.t_{end}, t_\infty)$ and add it to *gapsInRank*. The meaning of this last gap is that there is no time limitation to include any new task after the last task scheduled in a processor. Note that t_∞ represents a very high value.
5. Finally the algorithm returns the vector *gapsInRank* as an output.

When the task assignment algorithm has finished, the queue of tasks of each available processor is stored in *taskQueuePerCore* ordered ascendingly according to the tasks' start time. The only information that will be used by the task manager for running the tasks is the processors' ranks to which they have been assigned and the precedence order of the tasks that have been mapped to each processor. The real execution of each task will be started as soon as possible and it will last until the task is completed. The start time, end time and run time of each task that have been used by the assignment algorithm are just estimations which do not directly affect the real execution of the batch of tasks.

3.4 Theoretical case study of load balancing strategies

3.4.1 Design of the experiments

Modeling tasks

Before starting to design an appropriate theoretical case study in order to prove the validity of the proposed load balancing strategies, let us remember the key hypotheses that were formulated in section 3.2.4 of this chapter.

- Every task involves an arbitrarily divisible computational load.
- Every task is moldable, i.e. the number of processors assigned to the task cannot be modified after the execution of the task has started.
- The computational load imbalance that may arise when executing a batch of tasks in a parallel system can have 2 causes:
  - The heterogeneity in tasks' run times depending on their input variables.
  - The inappropriate selection of the ratio (no. individuals / no. processor groups).

The first step for designing the case study consists on identifying which features of the tasks have to be modeled. Since the goal of the experiments is to test the goodness of the load balance achieved by the proposed algorithms, the only relevant feature of the tasks is their completion time, which depends on i) the input variables of the task, and ii) the number of processors assigned for the execution of the task. Other tasks' characteristics like their internal computation/communication ratio or scalability have an effect on the load balance through the task's completion time, being this the only interface with the main optimizer.

Consequently, the simplest way of modeling a task is the use of a sleep application. Such a task does nothing but waiting until a specified amount of time has gone by and may be built by means of 2 methods: i) using the Linux sleep command and ii) manually using 2 variables for storing the system's time. The latter option involves creating 2 time variables, storing in one of them the task's start time (wall clock time), storing in the other one the wall clock time at every instant after the execution of the task started and comparing successively both time readings. When the difference between them reaches the specified value, the execution of the task is stopped. The second option has been chosen due to portability reasons so as to avoid the need of the sleep command provided by the operating system.

A single node holding 32 processors and which belongs to an experimental HPC platform has been used for executing the whole theoretical case study. Measurements are not affected by different processor speeds in this way and all processors have access to a common shared-memory space. Moreover, the fact of reserving the whole node for a single user guarantees there is no interference with other users' jobs.

An additional issue is the modeling of errors in the tasks' completion time estimations. Provided that the database of task times contains enough information for the time the load balancing algorithms are applied, small time estimation errors are expected for most of the tasks even if bigger errors may occasionally take place. This behavior is modeled by means of a normal distribution centered in the task's real completion time and characterized by a certain standard deviation. The notation utilized for representing an error is $(\mu_{error}, \sigma_{error})$, where the average μ is always equal to zero and the standard deviation σ is a relative error with respect to the true value given as a fraction of one. In this theoretical case study, it is assumed that the true completion time of every task is always known without the need of carrying out any sampling, and it is possible to insert a bounded estimation error when desired.

The implementation of the mentioned task characteristics has been done as follows. The utilized sleep application (let us call it *performanceCase*) calculates a completion time depending on the value of the optimization variable and on the number of processors assigned to it. When the task is run, its real completion time matches exactly this time. However, when the task is asked to provide a run time prediction for scheduling purposes, the real completion time is modified by applying the normal error distribution and the result is sent to the task scheduling algorithm. The main variables and the algorithm are detailed hereafter:

- One optimization variable (*optVar*): It is a real variable ranging in [0,1].
- A communicator (*taskWorld*): It is an MPI communicator shared by the processors in charge of executing the task.
- Number of assigned processors (*nProc*): Number of processors assigned for the task's execution.
- A time range $[t_{min}, t_{max}]$: The lower and upper bounds for the task's completion time if it is executed in a single processor. If the value of the optimization variable is 0, it will take t_{min} seconds to run the task in 1 processor. If the value of the optimization variable is 1, the task's simulation will take t_{max} seconds. For any

other value in the interval $[t_{min}, t_{max}]$, the task's completion time is calculated by means of linear interpolation.

- Time estimation error $(\mu_{error}, \sigma_{error})$: Average and standard deviation of the normal distribution which models the user defined time estimation error.

Algorithm G *Execution of performanceCase*

---

1. **IF** (processor is the root of the task)
   (a) Calculate the task time (see *Algorithm G.1*).
   (b) Sleep during the time calculated in the previous step.
   (c) Set the value of the objective function to be equal to *optVar*. Note that the fitness value is irrelevant for the case study to be carried out.
   (d) Send the fitness value to all other processors.
2. **ELSE**
   (a) Receive the fitness value from the root.

Algorithm G.1 *Calculate the task time*

---

1. Base time = $t_{min} + \frac{(t_{max} - t_{min})}{(1-0)} (optVar - 0)$
2. Calculate the parallel speedup, which is obtained using a function provided by the user and depends on the number of processors assigned to the task.
3. Real task time = base time / speedup.
4. **IF** (a task time prediction is wanted)

(a) The time estimation error ($\mu_{error}, \sigma_{error}$) is applied to the real task time in order to get the task time prediction. This is done by first calculating an average and a standard deviation ($\mu_{pred}, \sigma_{pred}$) for the predicted task time, i.e. μ_{pred} = (real task time + μ_{error} · real task time) and σ_{pred} = (σ_{error} · real task time), and finally by requesting to the random number generator of *performanceCase* to create a task time prediction according to a normal distribution defined by ($\mu_{pred}, \sigma_{pred}$).

Metrics

Another requirement for performing a case study is to find an appropriate set of metrics able to evaluate the quality of the results provided by the proposed load balancing algorithms. A list containing 6 metrics which measure several aspects of the execution of a batch of tasks is presented hereafter:

- ***Evaluation time:*** It is the average fraction of the makespan of the batch of tasks spent by the processors executing tasks.
- ***Scheduling time:*** It is the average fraction of the makespan of the batch of tasks spent by the processors running the task scheduling algorithm. Note that when the first task scheduler is run at the beginning of the batch, the time spent for obtaining the schedule is counted as scheduling time in every processor.
- ***Idle time:*** It is the average fraction of the makespan of the batch of tasks spent by the processors performing neither evaluation nor scheduling operations and is caused mainly by the following 3 synchronization delays: i) a processor finished all its assigned tasks and the batch of tasks is exhausted, so it has to wait until every other processor completes its assigned tasks; ii) a processor is ready to start its next task, but it has to wait because the task requires parallel execution and some other processors are not available yet; iii) a processor needs to access the task schedule in order to get its next task, but the access to shared-memory has been locked by some other processor and the first processor has to wait until it is unlocked (note that this synchronization delay can only take place when the dynamic task manager is used).
- ***Makespan of the batch of tasks:*** It is the wall clock time required for completing the execution of the batch of tasks. This is the most important metric when evaluating the parallel efficiency of any application [30] as it accounts not only for

the computational work, but also for any contributions arising from synchronization, communication and I/O. Thus, the makespan is used as the reference metric for evaluating the overall quality of the proposed task management algorithms.

- ***Nondimensionalized makespan of the batch of tasks:*** It is calculated as the makespan of the batch of tasks nondimensionalized by the average computational time required for the completion of a task. This metric is only representative for tasks with linear parallel scalability and can be useful for evaluating the overall quality of the proposed task management algorithms when batches with the same number of tasks but different average task completion times are compared.
- ***Idle Time Coefficient (ITC):*** It measures the quality of the computational load balance achieved by the task scheduling algorithm, but it is not suitable for being used with the dynamic task manager and is only representative for tasks with linear parallel scalability. The ITC is calculated as the overall idle computational time when executing a batch of tasks nondimensionalized by the average computational time required for the completion of a task. It represents how staggered the profile of a task schedule is (see Fig. 3.13) independently from the number of tasks in the batch or their average completion time. On one hand, the use of this metric together with the dynamic task manager is discouraged, because in that case not every idle time span is due to an improvable task schedule. On the other hand, reduction of idle time does not necessarily involve that a good task schedule has been obtained in case tasks do not scale linearly.

Simulated sets of experiments

The parametrization options differ depending on the selected task manager and are shown in the list hereafter:

- Self-scheduling task manager: the number of processors assigned per task.
- Static task manager: the lower and upper bounds defining the allowed number of processors per task, and the limitation of the scheduling time.
- Dynamic task manager: the lower and upper bounds defining the allowed number of processors per task, the limitation of the scheduling time, and the number of runs of the scheduling algorithm per batch of tasks.

Taking into account all the above considerations and the characteristics of the implemented load balancing strategies, 3 sets of experiments have been carried out so as to test the performance of the proposed load balancing strategies:

1. Short cases with linear parallel scalability: This set involves the execution of tasks requiring an average computational time of 300 seconds.

2. Long cases with linear parallel scalability: This set involves the execution of tasks requiring an average computational time of 1500 seconds and is aimed at proving that for sufficiently large makespans, the scheduling time can be enlarged and the load balance of the batch of tasks improved.

3. Long cases with non-linear parallel scalability: This set involves the execution of tasks requiring an average computational time of 1500 seconds and is aimed at testing the effect of non-linear cases on the task scheduling algorithm.

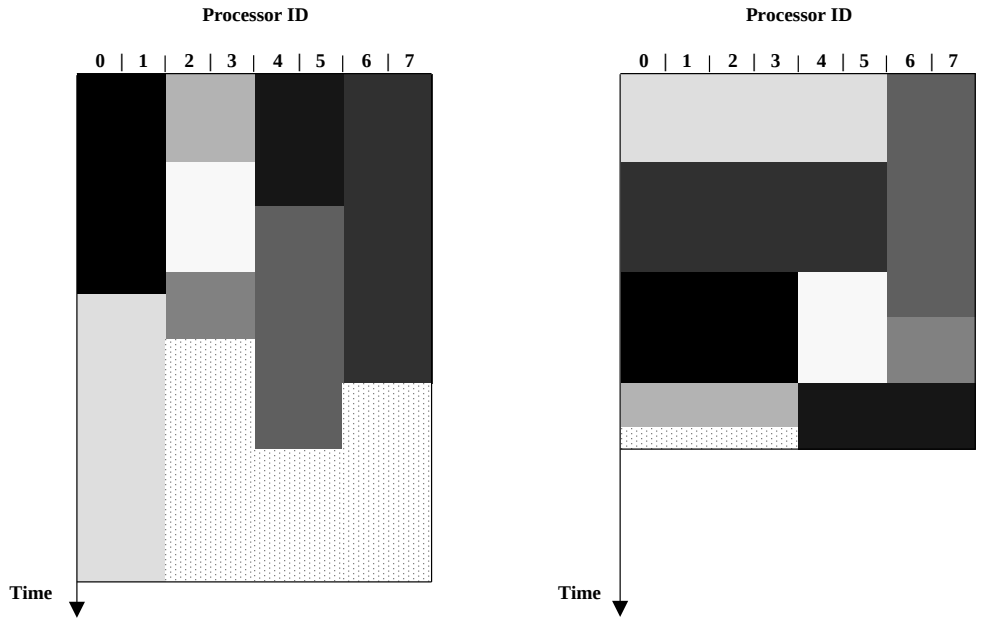


Figure 3.13: Representation of the idle time (dotted space) in two different task schedules proposed for the same batch of tasks. It can be seen that the task schedule on the right has lower idle time and makespan values than the one on the left. Every task scales linearly and the scheduling time required by the utilized static task manager has not been represented.

Every set of experiments has been run using the genetic algorithm implemented in Chapter 2 of this Doctoral Thesis and keeping the local search method inactive. But before analyzing the obtained results, let us have a look at the common characteristics of these three case studies:

- Size of the batch of tasks: Batches with 20, 24, 32, 34 and 60 tasks have been simulated, because they represent a sufficient number for being run in 32 processors and allow evaluating the effect of inappropriate ratios (no. individuals / no. processor groups).
- Scalability of the tasks: The use of tasks with linear parallel performance simplifies the evaluation of the quality of the implemented load balancing strategies, because the only loss of parallel efficiency may be caused by the optimizer. This means that the optimal task schedule is known to be the one which achieves 100% of CPU utilization (0% idle time) or an ITC equal to zero. In case of having tasks with non-linear scalability, an additional loss of parallel efficiency occurs when the task is executed in multiple processors and it is impossible to know a priori which the optimal task schedule is.
- Computational time of the tasks: Depending on the optimization variable provided by a uniform random number generator and defined in [0,1], each task is assigned a computational time in the bounded interval $[t_{min}, t_{max}]$. This interval is tuned so that the extremes account for a 0%, 20% and 50% variation over the desired average computational time, i.e. 300 seconds for short cases and 1500 seconds for long cases. For example, a case with an average computational time of 300 seconds and a maximum variation of 50% is bounded in the interval [150, 450] (in seconds).
- Number of generations: Each test involves the simulation of 10 batches of tasks (i.e. 10 generations) and averaging the aforementioned metrics.
- Random number generators: On one hand, the same seed has always been used for the random number generator in charge of creating and modifying the tasks, what means that if tests with the same size of the batch of tasks are compared, the optimization variables of the tasks of each generation will correspond exactly with each other. On the other hand, the results obtained by the task scheduling algorithm and the algorithm in charge of introducing time estimation errors are not repeatable. Nevertheless, this is not necessary as they are studied by means of averaged metrics.

- Genetic operators: Only the random mutation operator has been utilized for creating values for the optimization variable in the range [0,1] and with an application probability of 100%. The result is that a completely new batch of tasks is obtained in each generation, keeping no relationship with the previous generation. The reason of using this operator is that the usual tendency of optimizers is to destroy the population's diversity, being easier to balance a batch of similar individuals. However, keeping the diversity level is necessary so that averaging the metrics obtained in different generations makes sense.

3.4.2 Short cases with linear scalability

The first step has been the simulation of short cases in order to prove the existence of the aforementioned load balancing problem when the self-scheduling task manager is used. The obtained results are shown in Figs. 3.14, 3.15 and 3.16.

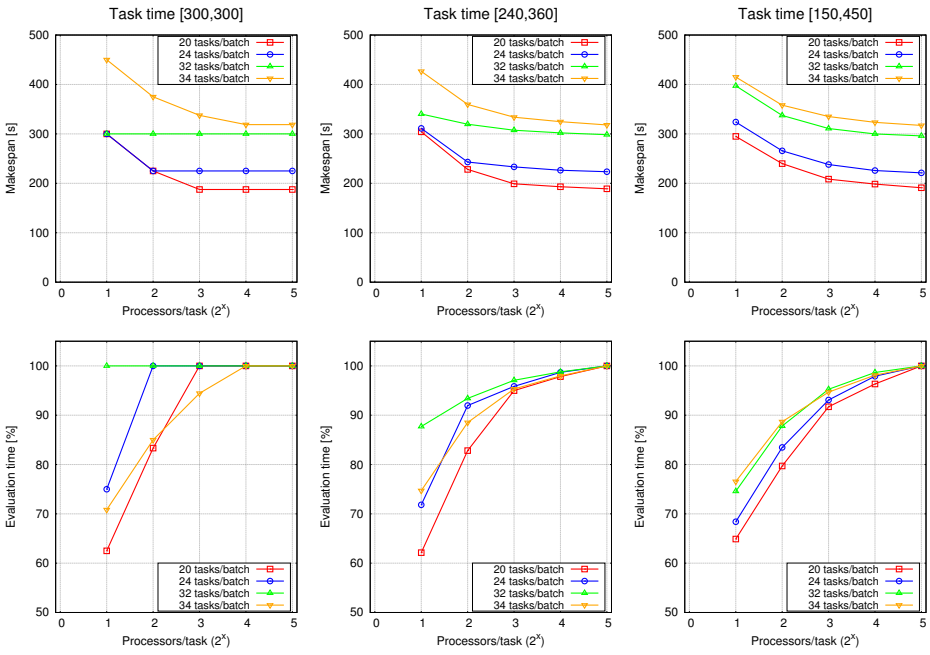


Figure 3.14: Comparison of the average makespan (in seconds) and evaluation time (percentage of the overall makespan) for batches of short tasks with linear scalability balanced by the self-scheduling task manager.

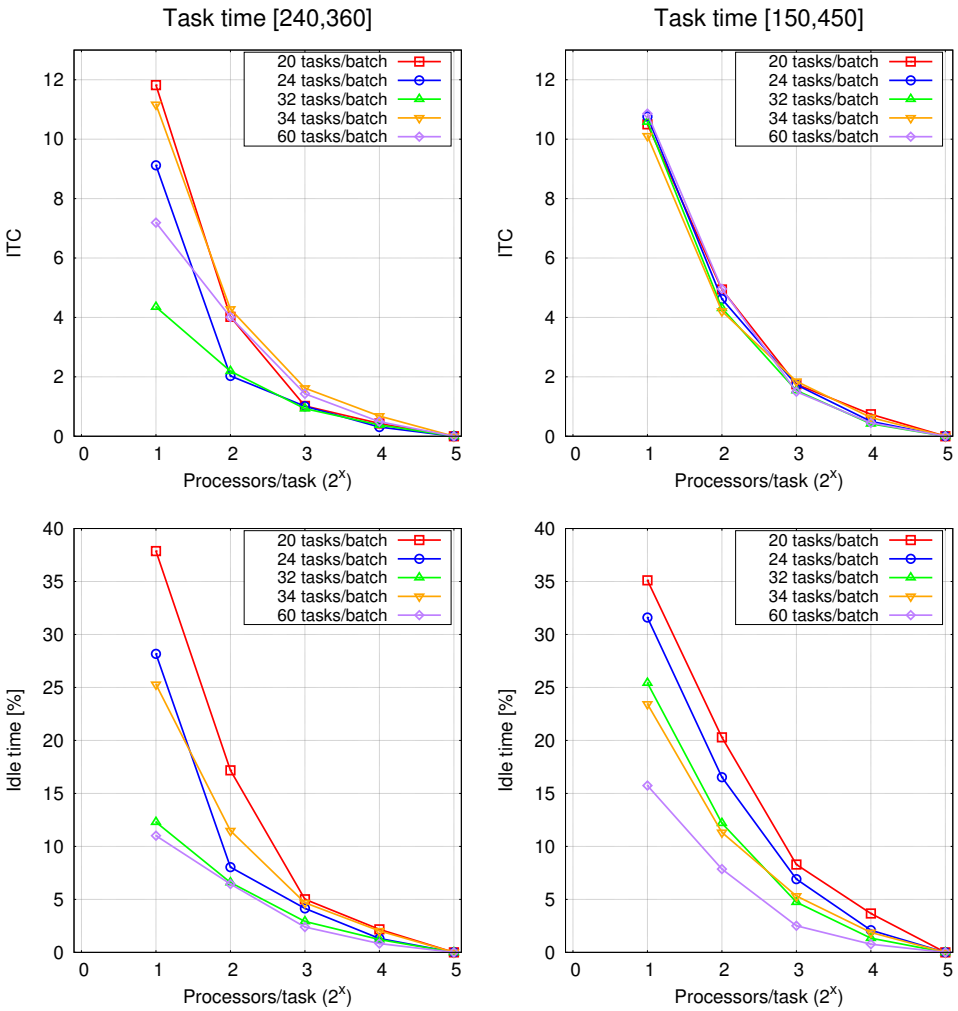


Figure 3.15: Comparison of the average Idle Time Coefficient (ITC) and idle time (percentage of the overall makespan) of batches of short tasks with linear scalability balanced by the self-scheduling task manager.

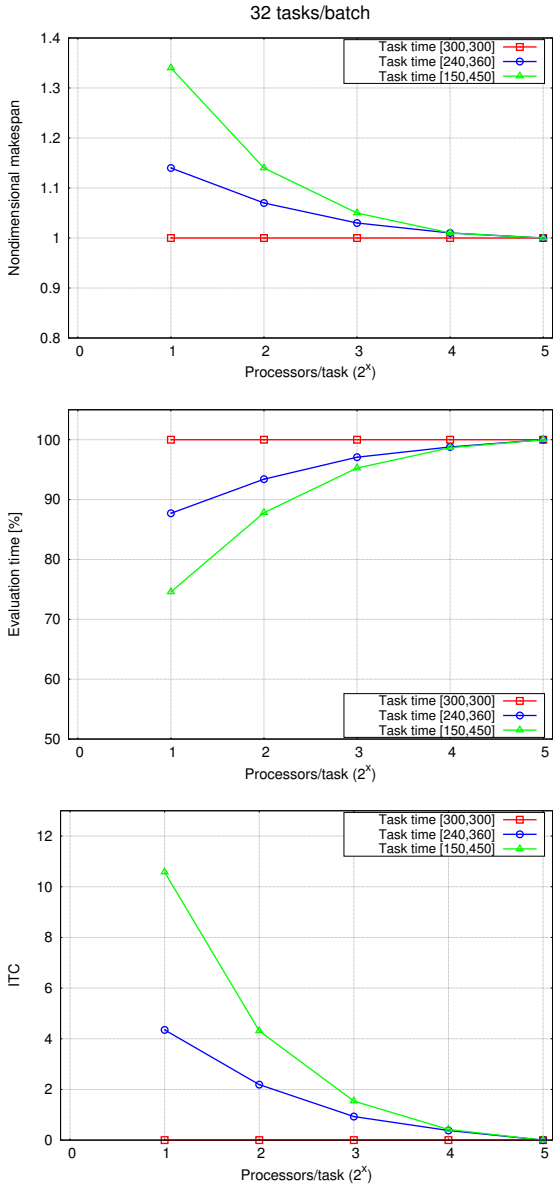


Figure 3.16: Comparison of the average nondimensional makespan, evaluation time (percentage of the overall makespan) and Idle Time Coefficient (ITC) of batches of short tasks with linear scalability balanced by the self-scheduling task manager.

It can be seen in Fig. 3.14 that 100% parallel efficiency (evaluation time percentage) is always achieved provided that every task is run using the 32 available processors, reaching the minimum value for the makespan. But let us observe the results obtained when every task requires a constant computational time of 300 seconds. If several tasks are executed simultaneously and the ratio (no. individuals / no. processor groups) is inappropriately selected, loss of parallel efficiency takes place leading to an increase of the average makespan. Note, for instance, that the same makespan was obtained when simulating 20 and 24 tasks assigning 4 processors per task. The heterogeneity in tasks' run times has a similar effect although the (no. individuals / no. processor groups) ratio is properly selected by the user, being impossible to avoid the loss of parallel efficiency caused by the computational load imbalance.

This load imbalance can also be measured by means of the idle time percentage and the Idle Time Coefficient (ITC), as it is shown in Fig. 3.15. The fewer the tasks contained in the batch are, the greater the idle time percentage is but no meaningful change arises regarding the ITC. This behavior is emphasized as the heterogeneity in tasks' run times increases, e.g. run times in the range [150,450]. The reason is that for a given number of processors assigned per task and the greater the size of the batch of tasks is, the idle computational time accounts for a lower percentage over the overall makespan of the batch. Thus, the ITC is the preferred metric for evaluating the quality of schedules as it only takes the staggering at the end of the execution of the batch into account.

Batches of 20, 24, 32, 34 and 60 tasks have been used so far in order to evaluate the negative impact of the computational load imbalance in a general case. However, the user is expected to configure the self-scheduling task manager so that an appropriate ratio (no. individuals / no. processor groups) is used in order to try to avoid this load imbalance. A batch composed by 32 tasks has the minimum size that fulfills this condition for any number of processors that can be assigned per task (i.e. 1,2,4,8,16 or 32) as it was proved by obtaining parallel efficiencies (i.e. evaluation time percentages) of 100% in case of having constant tasks' computational times. Hence, the batch of 32 tasks is taken as the main reference for further comparisons.

It is clearly depicted in Fig. 3.16 that the greater the heterogeneity in tasks' run times, the greater the loss of parallel efficiency. Consequently, both the average makespan of the batch of tasks and the ITC have higher values. Although the average computational task time is theoretically independent from the task times' heterogeneity in the batch, the reality is that this cannot be guaranteed averaging the results obtained in just 10 generations. Hence, this distorting factor is eliminated by

nondimensionalizing the makespan with the average computational task time.

The load balancing problem that arises when the self-scheduling task manager is used has been described up to this point. The static task manager was introduced as an improved load balancing algorithm, provided that it was possible to perform perfect task time estimations. The results obtained by this algorithm for the same short cases are presented hereafter. The method was configured to allow assigning 2^x processors per task, being x in the range [0,5], and the scheduling time was limited to 1%, 5% and 10% of the previous generation's makespan.

Several observations can be made regarding the results shown in Fig. 3.17. In first place, they prove that the greater the scheduling time is, the lower the obtained ITC and makespan without considering the scheduling time are, i.e. a better load balance is achieved. However, the best overall solutions (smallest makespan taking into account the scheduling time) were obtained when the scheduling time was limited to a 1% value of the previous generation's makespan most of the times. This fact indicates that the scheduling overhead was too high in comparison to one generation's makespan when 5% and 10% scheduling time limitations were used. In second place, note that the smaller the size of the batch of tasks is, the better the obtained ITC values are most of the times. The reason is that the size of the combinatorial problem to be solved by the scheduling algorithm is smaller in these cases, being the convergence speed to the optimal solution faster. Finally, it can be seen that the scheduling algorithm was able to find the optimal solution (ITC equal to zero) many times when the tasks' computational times were constant. However, finding the optimal solution (obtained by assigning 32 processors per task) for heterogeneous tasks' run times seems to be more difficult.

The results obtained for the reference batch size of 32 tasks are represented in Fig. 3.18. Just for mentioning a detail, note that for the scheduling time limitation of 5% a similar nondimensional makespan without considering the scheduling time was obtained for task times in the ranges [240,360] and [150,450]. However, the nondimensional makespan of the latter case is lower, which means that less scheduling time than in the first case was required.

The results provided by the self-scheduling and the static task managers for the batch size of 32 tasks are overlapped in Fig. 3.19. On one hand, results corresponding to the best makespan including the scheduling time obtained by the static task manager are represented. On the other hand, results corresponding to the best makespan without considering the scheduling overhead are included. Note that they are independent from each other (i.e. in a certain graph, the makespan with and without the scheduling time do not correspond to the same load distribution), as achieving the best load balance

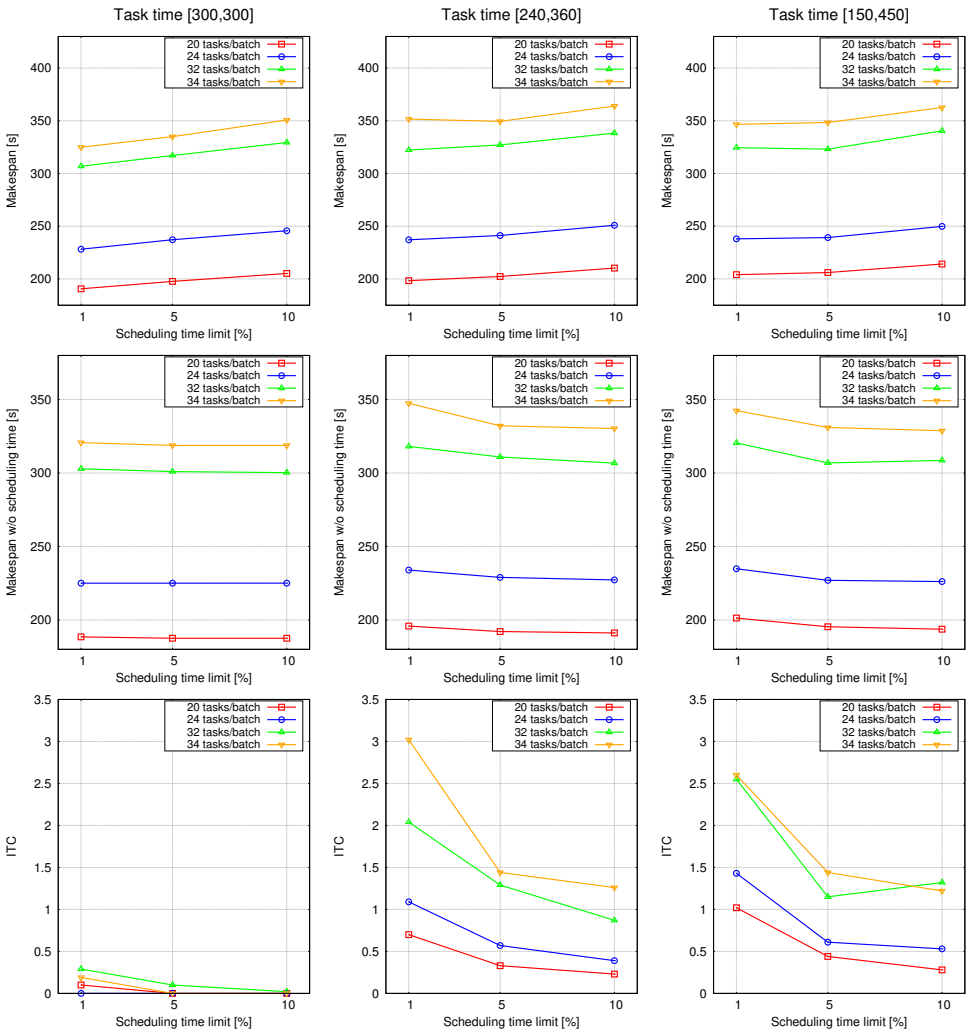


Figure 3.17: Comparison of the average makespan (in seconds), makespan without considering the scheduling time (in seconds) and Idle Time Coefficient (ITC) for batches of short tasks with linear scalability balanced by the static task manager.

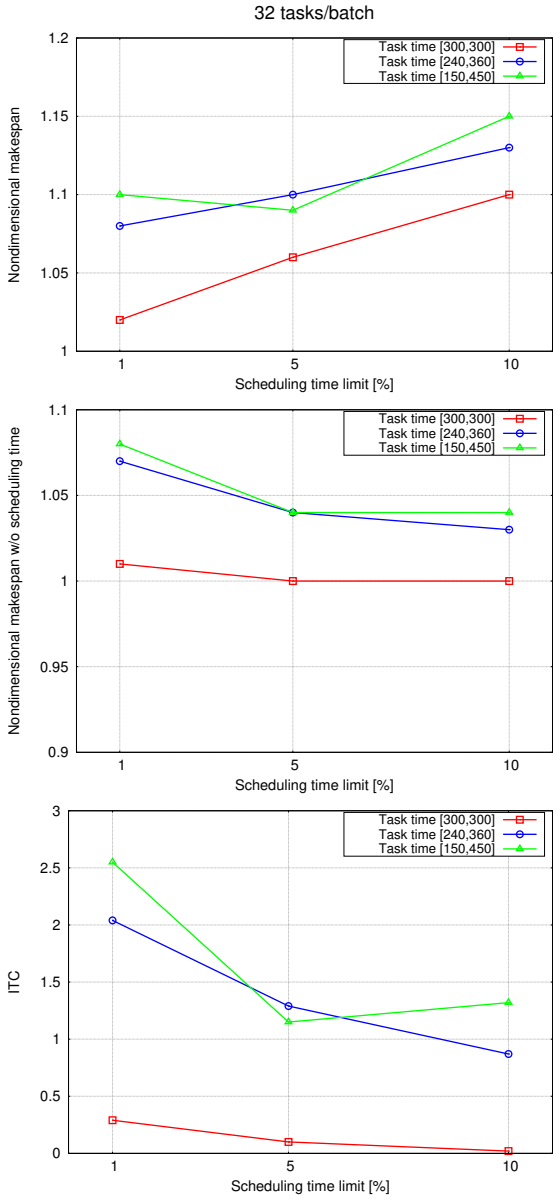


Figure 3.18: Comparison of the average nondimensional makespan, nondimensional makespan without considering the scheduling time and Idle Time Coefficient (ITC) for batches of 32 short tasks with linear scalability balanced by the static task manager.

may involve an important scheduling overhead which does not lead to the best overall makespan. The results provided by the static task manager have been represented as horizontal lines, even if each line corresponds to a single run of the optimizer (i.e. the evaluation of 10 batches of tasks). The aim is to ease the comparison with the results obtained by the self-scheduling task manager by making visible the intersection between both lines.

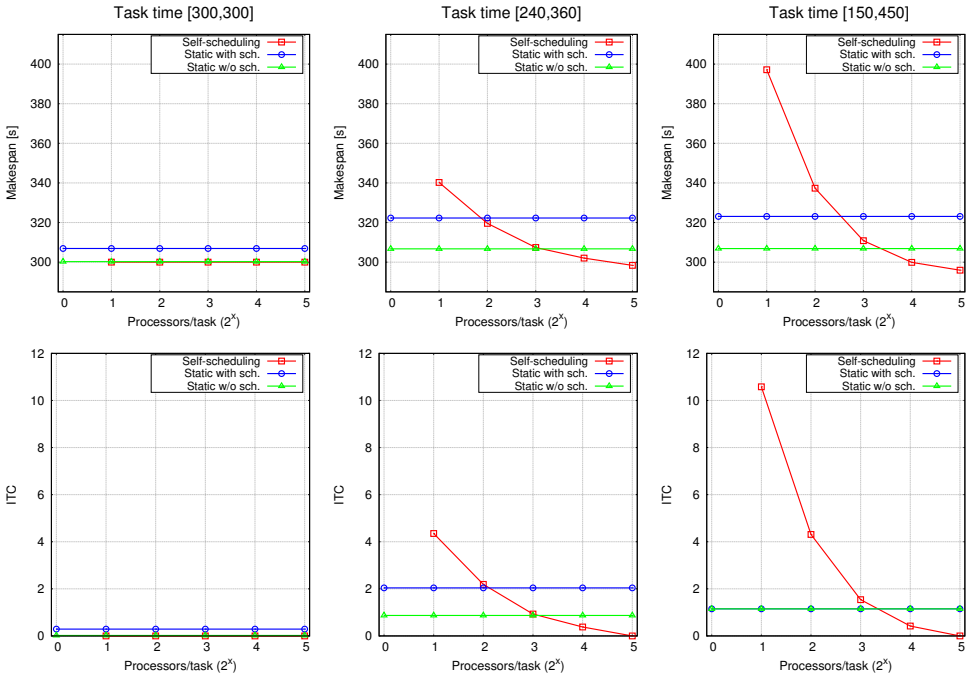


Figure 3.19: Comparison of the average makespan (in seconds) and Idle Time Coefficient (ITC) for batches of 32 short tasks with linear scalability obtained by the self-scheduling and static task managers.

Although an optimal solution for the load balancing problem involving a 100% parallel efficiency existed in every case, the scheduling algorithm was able to find it only when the tasks' computational time was constant. This means that either the allowed scheduling time or the scheduling algorithm's convergence speed should be increased so as to be able to find the optimal task distribution for batches with heterogeneous tasks' times. Regarding this kind of tasks and if the scheduling overhead

is not taken into account, it can be seen that the results provided by the static task manager outperform those obtained by the self-scheduling algorithm when 4 tasks were executed simultaneously, i.e. $2^3 = 8$ processors were assigned per task. In case the scheduling overhead is considered, results provided by the static task manager get close to those obtained by the self-scheduling task manager when 8 individuals were executed simultaneously.

The trend observed in the results provided by the self-scheduling task manager is that the more tasks are executed simultaneously, the lower the parallel efficiency is. Moreover, this task manager is unable to handle heterogeneity in tasks' run times properly. The static task manager has demonstrated to be clearly beneficial in such situations, being able to handle satisfactorily tasks with variable run times (see Fig. 3.19).

Although the ability of estimating task times accurately has been assumed so far, the presence of estimation errors may be common. The negative impact of such errors on the static task manager is studied in Fig. 3.20. The average makespan of a batch of tasks without considering the scheduling time is the metric that has been chosen to quantify the degradation of the load balance. As it can be seen, the common trend is that the makespan and the ITC increase as the tasks' completion time estimation errors increase. In case of having estimation errors characterized by a standard deviation between 0-10%, the degradation of the load balance is noticeable but the obtained results are considered acceptable. However, the results provided by the static task manager in case of having errors with a standard deviation of up to 30% are worse than those achieved by the self-scheduling algorithm.

The dynamic task manager is better suited for handling task time estimation errors than the static task manager, and it has been configured as follows:

- The scheduling algorithm has been run twice per batch of tasks.
- It was allowed to assign 2^x processors per task, being x in the range [0,5].
- The scheduling time was limited to 1%, 5% and 10% of the previous generation's makespan.

The results provided by the self-scheduling, static and dynamic task managers for the batch size of 32 tasks are overlapped in Fig. 3.21. In the case of the static and dynamic algorithms, the results corresponding to the best obtained makespan including the scheduling time are represented. Note that this is the only suitable metric in order to compare both task managers, because the scheduling overhead must be taken

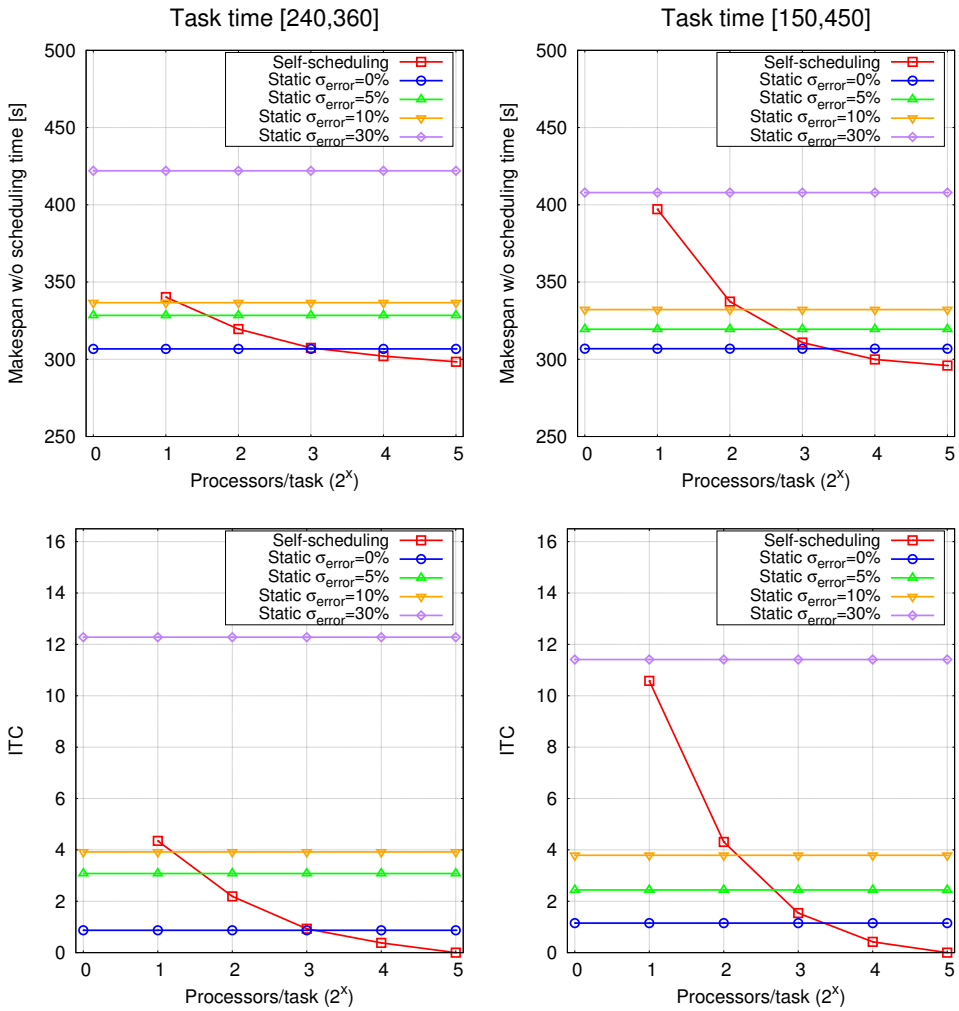


Figure 3.20: Comparison of the average makespan without considering the scheduling time (in seconds) and Idle Time Coefficient (ITC) for batches of 32 short tasks with linear scalability obtained by the self-scheduling and static task managers and subject to task time estimation errors.

into account. The results provided by the static and dynamic algorithms have been represented as horizontal lines, even if each line corresponds to a single run of the optimizer (i.e. the evaluation of 10 batches of tasks). The aim is to ease the comparison with the results obtained by the self-scheduling task manager by making visible the intersection between the lines.

On one hand, in the simulations carried out with tasks requiring a computational time in the range [240,360] the static task manager outperformed the dynamic one in case of having task time estimation errors characterized by standard deviations of 0% and 10%. Nevertheless, the dynamic task manager performed better than the static one in case of having 5% and 30% errors. On the other hand, in simulations with tasks requiring a computational time in the range [150,450] the dynamic task manager always outperformed the static one. These results suggest that the use of the dynamic algorithm is more beneficial the higher the heterogeneity in tasks' run times is and the greater the task time estimation errors are.

Several conclusions may be extracted from the presented case study. First, the task management and scheduling algorithms have proved to be able to successfully balance the computational load of batches of tasks with heterogeneous completion times. The main drawbacks of the proposed algorithms are the scheduling overhead and handling the task time estimation errors. Regardless of the utilized task manager, the best makespans have been obtained for scheduling time limitations of 1% and 5%, what suggests that even better makespans might had been obtained by allowing lower time limitations.

Note also that the same scheduling algorithm is utilized by both the static and dynamic task managers, although it is called more often and with updated information by the latter. This allows getting a better balanced task schedule at the cost of a higher scheduling time. The positive impact caused by an improvement in the convergence speed of the scheduling algorithm would be more noticeable in the dynamic task manager, whose results might be improved with respect to the static task manager.

Finally, it is observed that if tasks with a greater average completion time were simulated, a certain scheduling time would represent a lower percentage over the makespan of the batch of tasks compared to the results obtained for short cases. Hence, increasing the scheduling time in order to get better task distributions would be affordable. This has been the motivation for studying cases which involve a longer computational time in the following sections.

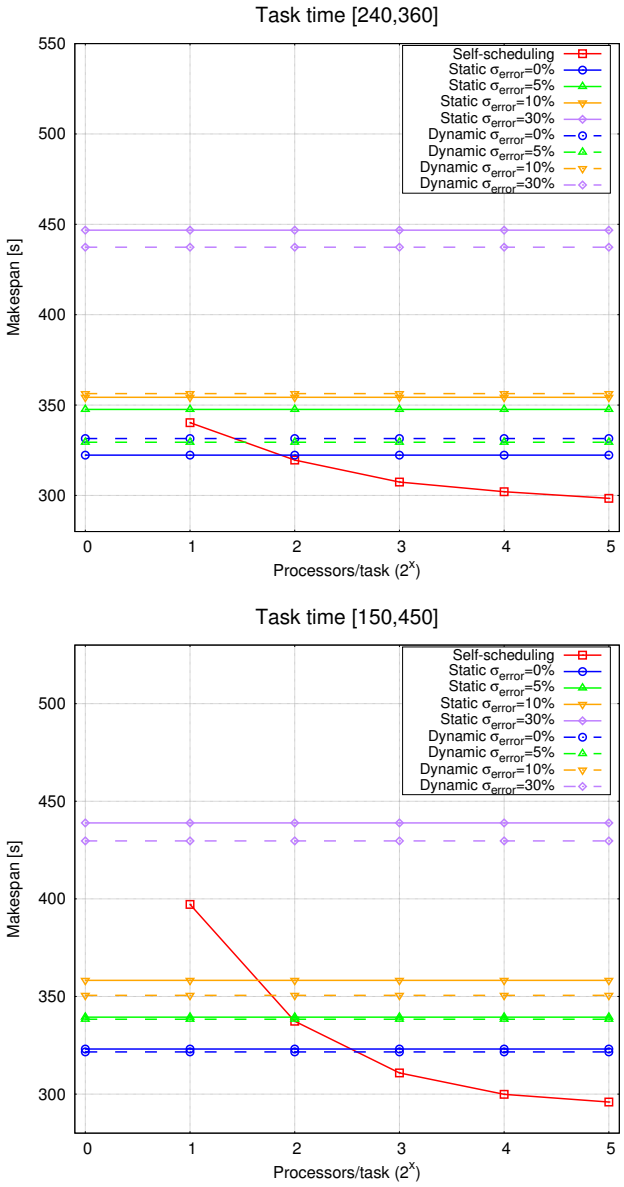


Figure 3.21: Comparison of the average makespan (in seconds) for batches of 32 short tasks with linear scalability obtained by the self-scheduling, static and dynamic task managers and subject to task time estimation errors.

3.4.3 Long cases with linear scalability

The simulation of short sleep tasks involving an average computational time of 300 seconds is not representative of the real target application of the load balancing algorithms proposed so far, i.e. the optimization of CFD & HT models. Therefore, a second case study using long sleep jobs requiring an average computational time of 1500 seconds has been carried out. The increase in the average computational task time is beneficial for the task management algorithm, because a certain scheduling time represents a lower percentage over the average makespan of the batch of tasks if compared to the results obtained for short cases. Simulations with task times in the ranges [1200,1800] and [750,2250] have been carried out, keeping the same heterogeneity that was used for short cases. Additionally the range [300,2700] has been considered, which involves a maximum variation of 80% with respect to the tasks' average computational time. With the aim of reducing the number of cases to be simulated, only the reference batch size of 32 tasks has been analyzed.

Three changes have been introduced in the configuration of the static and dynamic task managers:

- The use of 2^x processors per task was allowed, being x in the range [0,4] instead of in the range [0,5]. The aim of this modification is to reduce the size of the scheduling combinatorial problem in the hope of obtaining better results. It was decided to suppress the possibility of assigning every processor to a single task because this is unlikely to happen in a real application, making the use of a scheduling algorithm no sense in such a situation.
- The scheduling time is limited to 1%, 2% and 3% of the previous generation's makespan.
- The dynamic task manager may run the scheduling algorithm 2 or 3 times per batch of tasks (both possibilities have been tested for every case).

First, the cases were simulated using the static task manager assuming perfect task time estimations. The obtained results are shown in Fig. 3.22. Although the average computational task time is theoretically independent from the task times' heterogeneity in the batch, the reality is that this cannot be guaranteed averaging the results obtained in 10 generations. Hence, this distorting factor is eliminated by nondimensionalizing the makespan with the average computational task time. The trend of the nondimensionalized makespan is that the greater the scheduling time, the

greater the makespan. However, the opposite behavior of the nondimensional makespan is observed if the scheduling time is not taken into consideration, being the obtained results better the greater the scheduling time is. This trend is shown by the ITC, as well.

It is concluded that the static task manager is able to properly balance the computational load and can provide a reduction of the makespan provided that an excessive scheduling overhead is avoided. A couple of time references are given next. On one hand, the best makespans (taking into account the scheduling time) for every task time heterogeneity were obtained in approximately 16 seconds. On the other hand, the best makespans without considering the scheduling time were obtained in approximately 46 seconds. Finally, and as it happened for short cases, it was noticed that the difficulty of balancing the computational load of the batch of tasks increases with the heterogeneity of tasks' computational times.

The results provided by the self-scheduling and the static task managers are overlapped in Fig. 3.23. On one hand, results corresponding to the best makespan including the scheduling time obtained by the static task manager are represented. On the other hand, results corresponding to the best makespan without considering the scheduling overhead are included. Note that they are independent from each other (i.e. in a certain graph, the makespan with and without the scheduling time do not correspond to the same load distribution), as achieving the best load balance may involve an important scheduling overhead which does not lead to the best overall makespan. The results provided by the static task manager have been represented as horizontal lines, even if each line corresponds to a single run of the optimizer (i.e. the evaluation of 10 batches of tasks). The aim is to ease the comparison with the results obtained by the self-scheduling task manager by making visible the intersection between both lines.

It can be seen that the makespans without considering the scheduling time achieved by the static task manager are very similar to those obtained by the self-scheduling algorithm when 16 processors were assigned for executing each task. Hence, the best known solution when 16 processors/task are used is being fairly well approached. Good results are also obtained if the scheduling time is included in the makespan, being the ones provided by the static task manager similar to those obtained by the self-scheduling algorithm when 8 processors were assigned per task. Note the increase in the makespans obtained by the self-scheduling task manager caused by the heterogeneity increase in tasks' run times. Using the static task manager would clearly help avoid such behaviors.

Although the ability of performing accurate task time estimations has been assumed so far, errors will unavoidably arise when handling batches of real tasks. The negative

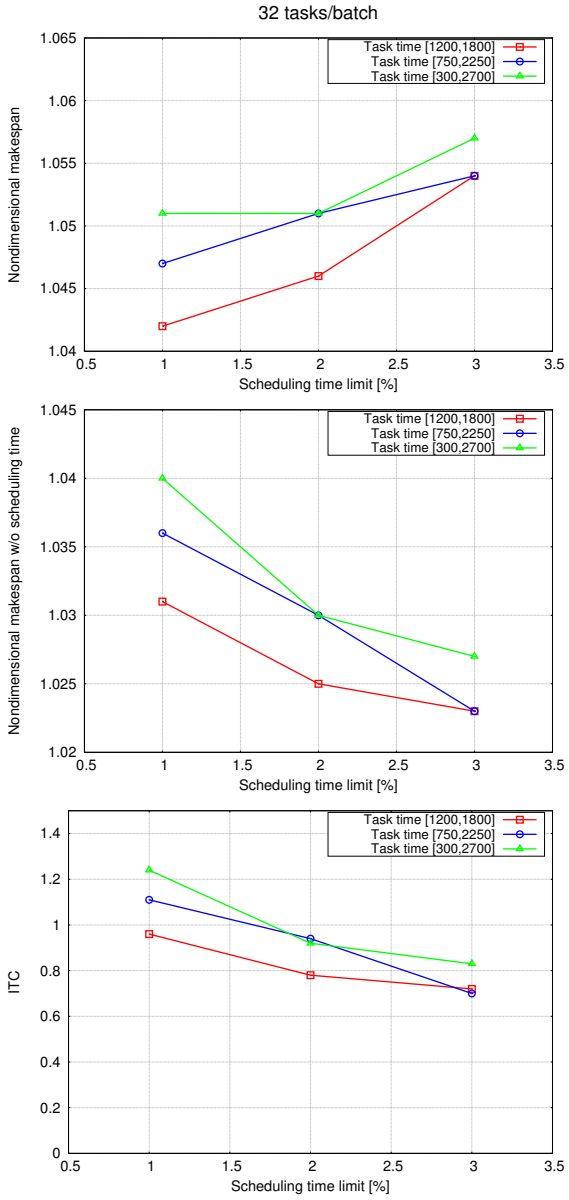


Figure 3.22: Comparison of the average nondimensional makespan, nondimensional makespan without considering the scheduling time and Idle Time Coefficient (ITC) for batches of 32 long tasks with linear scalability balanced by the static task manager.

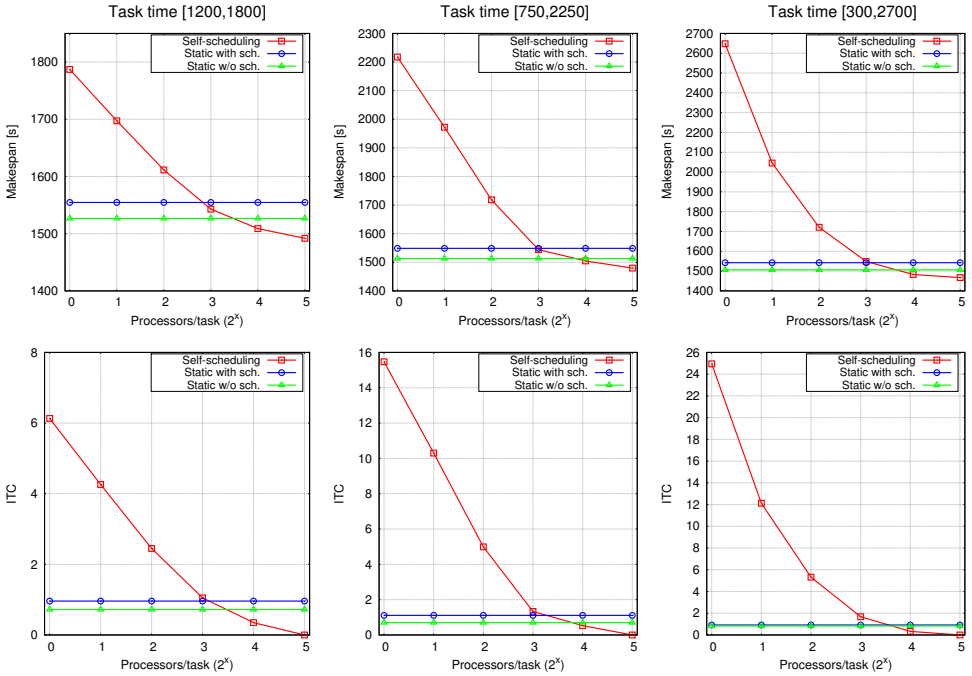


Figure 3.23: Comparison of the average makespan (in seconds) and Idle Time Coefficient (ITC) for batches of 32 long tasks with linear scalability obtained by the self-scheduling and static task managers.

impact of such errors on the results provided by the static task manager is studied in Fig. 3.24. The average makespan of a batch of tasks without considering the scheduling time has been the selected metric in order to quantify the degradation of the load balance. It is observed that an increase in the task time estimation errors always causes the degradation of the computational load balance. Nonetheless, the ITC has similar values for a given task time estimation error independently from the tasks' run time heterogeneity. Moreover, the makespans obtained by the self-scheduling task manager when 1 or 2 processors are assigned per task show a considerable load imbalance, especially for high levels of heterogeneity in tasks' run times. The static task manager proved its capacity to outperform these results even when estimation errors modeled by a standard deviation of 30% over the average task time were applied.

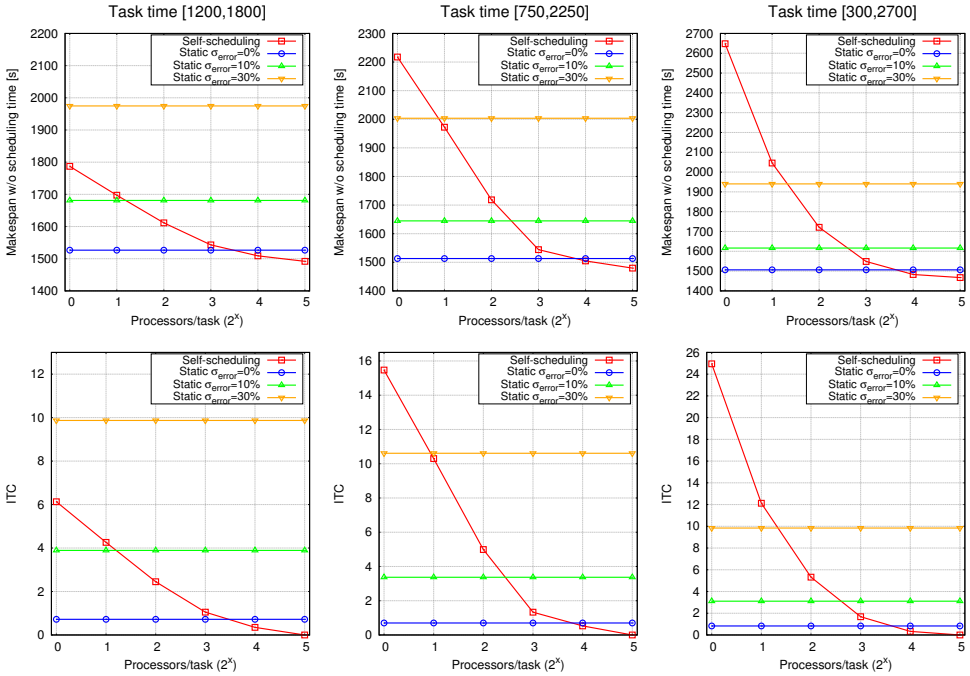


Figure 3.24: Comparison of the average makespan without considering the scheduling time (in seconds) and Idle Time Coefficient (ITC) for batches of 32 long tasks with linear scalability obtained by the self-scheduling and static task managers and subject to task time estimation errors.

Simulations of batches composed by short and long cases are compared in Fig. 3.25 with the aim of showing the static task manager's performance improvement as the tasks' average computational time is increased. Results obtained from simulating short tasks are placed on the left column, whereas the results coming from the simulation of long tasks are on the right column. These two kinds of tasks are characterized by requiring different average computational times, but the same heterogeneity with respect to the average time has been considered in both case studies, i.e. maximum variabilities of 20% (upper row) and 50% (lower row) with respect to the average computational time. Note that perfect task time estimations were assumed for every case.

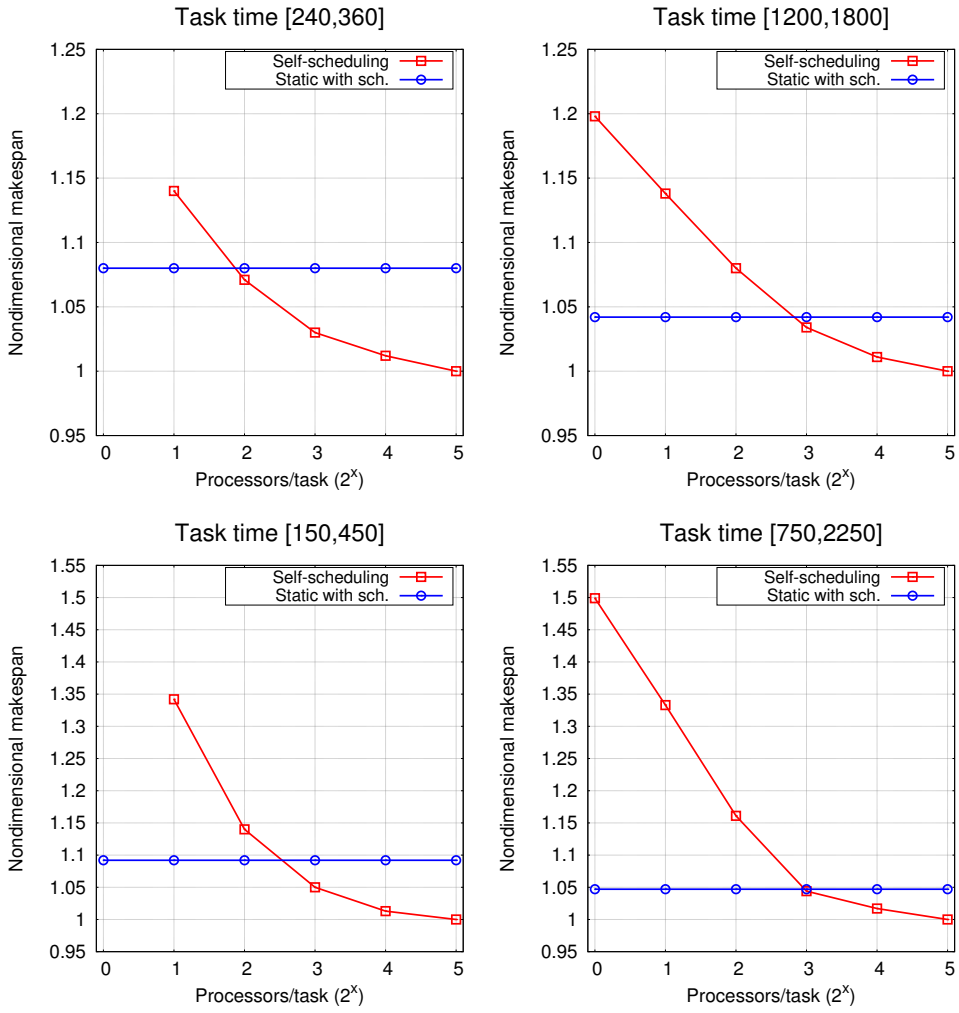


Figure 3.25: Comparison of the average nondimensional makespan for batches of 32 short and long tasks with linear scalability obtained by the self-scheduling and static task managers.

It can be seen that the quality of the results provided by the static task manager in relation to those obtained by the self-scheduling algorithm was better when long tasks were simulated (right column). The reason is that having tasks with a longer average computational time allows the scheduling algorithm to take more time for distributing the computational load properly, as well as reducing the relative scheduling overhead with respect to the average makespan of the batch of tasks. This effect is clearly visible in Fig. 3.25, where the same scale is used for representing the two graphs in each row. Note that the horizontal lines (average makespans obtained by the static task manager) on the right column are at a lower height than their analogous on the left column.

The dynamic task manager is better suited for handling task time estimation errors than the static task manager and for these experiments it has been configured so that the scheduling algorithm is run either twice or three times per batch of tasks. The results provided by the self-scheduling, static and dynamic task managers are overlapped in Fig. 3.26. In the case of the static and dynamic algorithms, the results corresponding to the best obtained makespan including the scheduling time are represented. Note that this is the only suitable metric in order to compare both task managers, because the scheduling overhead must be taken into account. The results provided by the static and dynamic algorithms have been represented as horizontal lines, even if each line corresponds to a single run of the optimizer (i.e. the evaluation of 10 batches of tasks). The aim is to ease the comparison with the results obtained by the self-scheduling task manager by making visible the intersection between the lines.

9 different cases were simulated, arising as combinations of 3 heterogeneity degrees in tasks' computational times and 3 task time estimation errors, trying several task manager configurations. The obtained results show that the dynamic task manager outperformed the static one in 7 of those 9 cases, proving the superiority of the dynamic algorithm with respect to the static algorithm.

Some observations regarding the dynamic task manager's configuration are introduced hereafter. On one hand, the best results in case of having perfect task time estimations were always provided by the configuration running the scheduling algorithm twice per batch of tasks with a scheduling time limitation of 1% over the previous generation's makespan. On the other hand, running the scheduling algorithm 3 times allowed obtaining the minimum makespan in case of having task time estimation errors. These errors were present in 6 of the 9 cases and a 2% scheduling time limitation provided the best results in 3 of them, a 1% limitation in 2 of them and a 3% limitation in one of them.

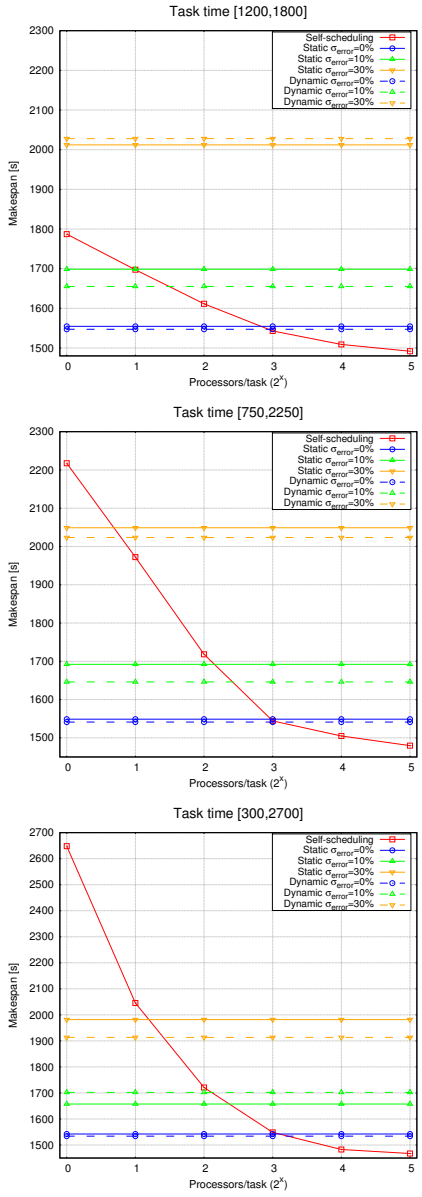


Figure 3.26: Comparison of the average makespan (in seconds) for batches of 32 long tasks with linear scalability obtained by the self-scheduling, static and dynamic task managers and subject to task time estimation errors.

These facts lead us to the following conclusions. First, it is advantageous to run the scheduling algorithm more than once per batch of tasks (twice according to the obtained results) in case of having perfect task time estimations. The reason is that the size of the second scheduling combinatorial problem is smaller than the first one, as some tasks have already been completed, what allows obtaining a better computational load distribution in a relatively short time. Second, it is advantageous to increase the number of executions of the scheduling algorithm per batch of tasks in the presence of task time estimation errors, because the chances for balancing the tasks' computational load using additional run time information are increased. Finally, note that the best results were often obtained when a 1% scheduling time limitation was used, what suggests that better makespans could have been reached by decreasing the scheduling time limitation even more.

According to the presented results, the proposed load balancing methods showed a good behavior when the standard deviation modeling task time estimation errors was bounded between 0% and 10%. However, higher errors had a considerable negative impact on the load balancing capacity of the task managers. It is reminded that the results for each case executed with a certain task manager's configuration were obtained by averaging 10 generations. This amount was considered sufficient for a preliminary study, but increasing the number of generations is necessary in order to increase the reliability of the reached conclusions.

3.4.4 Long cases with non-linear scalability

In this last case study the non-linear scalability of real tasks is taken into account by modeling the degradation of parallel performance taking place as they are assigned an increasing number of processors. The main problem when configuring the self-scheduling task manager consists on the impossibility of determining the optimal number of processors that should be assigned per task in order to minimize the makespan of the whole batch of tasks. On one hand, assigning few processors per task involves the execution of many tasks simultaneously, which is prone to generate computational load imbalance and the consequent loss of parallel performance. On the other hand, assigning many processors per task involves the execution of few tasks simultaneously, but it may lead to the degradation of the tasks' parallel efficiency. With the aim of simulating such a behavior, a sleep job with a fictitious parallel speedup described by the following function has been implemented: speedup = $-0.0005x^3 - 0.0025x^2 + x + 0.003$. Note that x is a natural number referring to the number of processors in which the task is executed.

Fig. 3.27 shows the speedup graph corresponding to the aforementioned mathematical expression.

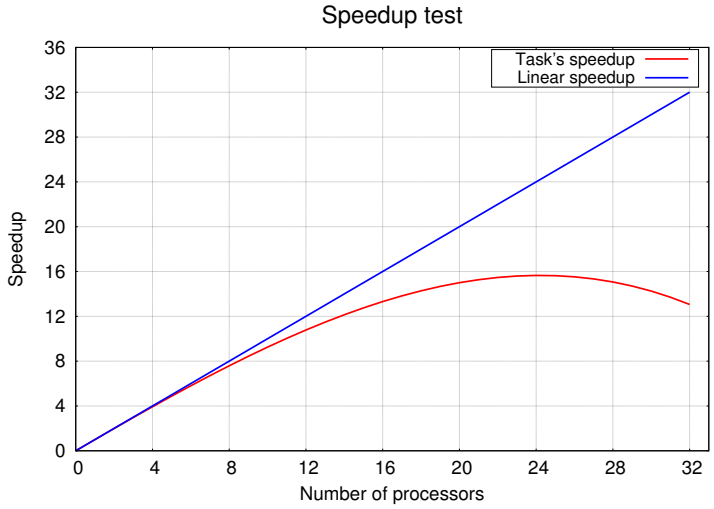


Figure 3.27: Graphical representation of a task's parallel speedup (speedup = $-0.0005x^3 - 0.0025x^2 + x + 0.003$) compared to linear speedup.

Nondimensionalized metrics, i.e. nondimensionalized makespan and Idle Time Coefficient (ITC), have not been used in this case study. The reason is that measuring a real-world non-linearly scalable task's computational time is not usually possible. Hence, the average makespan (in seconds) of a batch of tasks has been the only metric used for comparing the results obtained by the different task managers.

Simulations with task computational times in the ranges [1500,1500], [1200,1800], [750,2250] and [300,2700] have been carried out using the reference batch size of 32 tasks. The static and dynamic task managers have been configured as follows:

- The use of 2^x processors per task was allowed, being x in the range [0,5].
- The scheduling time is limited to 1%, 2% and 3% of the previous generation's makespan.
- The dynamic task manager may run the scheduling algorithm 2 or 3 times per batch of tasks (both possibilities have been tested for every case).

The results provided by the self-scheduling and the static task managers are overlapped in Fig. 3.28. On one hand, results corresponding to the best makespan including the scheduling time obtained by the static task manager are represented. On the other hand, results corresponding to the best makespan without considering the scheduling overhead are included. Note that they are independent from each other (i.e. in a certain graph, the makespan with and without the scheduling time do not correspond to the same load distribution), as achieving the best load balance may involve an important scheduling overhead which does not lead to the best overall makespan. The results provided by the static task manager have been represented as horizontal lines, even if each line corresponds to a single run of the optimizer (i.e. the evaluation of 10 batches of tasks). The aim is to ease the comparison with the results obtained by the self-scheduling task manager by making visible the intersection between both lines.

When tasks requiring a constant computational time of 1500 seconds were executed, the optimal makespan was achieved assigning a single processor per task. It is also observed that the highest loss of parallel efficiency took place as each task was executed in 32 processors, leading to a makespan which at least doubles the one obtained when 1 processor was assigned per task. Although the static task manager was unable to find the optimal task distribution, better results are expected when a more powerful task scheduling algorithm is implemented.

In the case of simulations run with heterogeneous tasks' computational times, the best results provided by the self-scheduling task manager have always been outperformed by the static task manager either the scheduling time was taken into consideration or not. Rigidity due to the fact that the same number of processors has to be assigned per task is the main drawback of the self-scheduling algorithm. Note also that the best parallel performance was reached in every case by assigning 8 processors per task. Makespan reductions for different configurations of the self-scheduling algorithm achieved by the static task manager are shown in Table 3.1, where only the best results obtained by the static task manager have been included.

Let us compare the average makespan of the batches of tasks taking into account the scheduling overhead. The parallel speedup of tasks (see Fig. 3.27) is the only information available when configuring the self-scheduling task manager. According to this, the user can decide between two options. The first one consists on maximizing the parallel efficiency of tasks (i.e. reducing the number of processors assigned per task) in the hope that this strategy will lead to minimize the makespan of the batches of tasks. However, if the results obtained by the self-scheduling task manager with a single processor assigned per task are compared to those provided by the static task

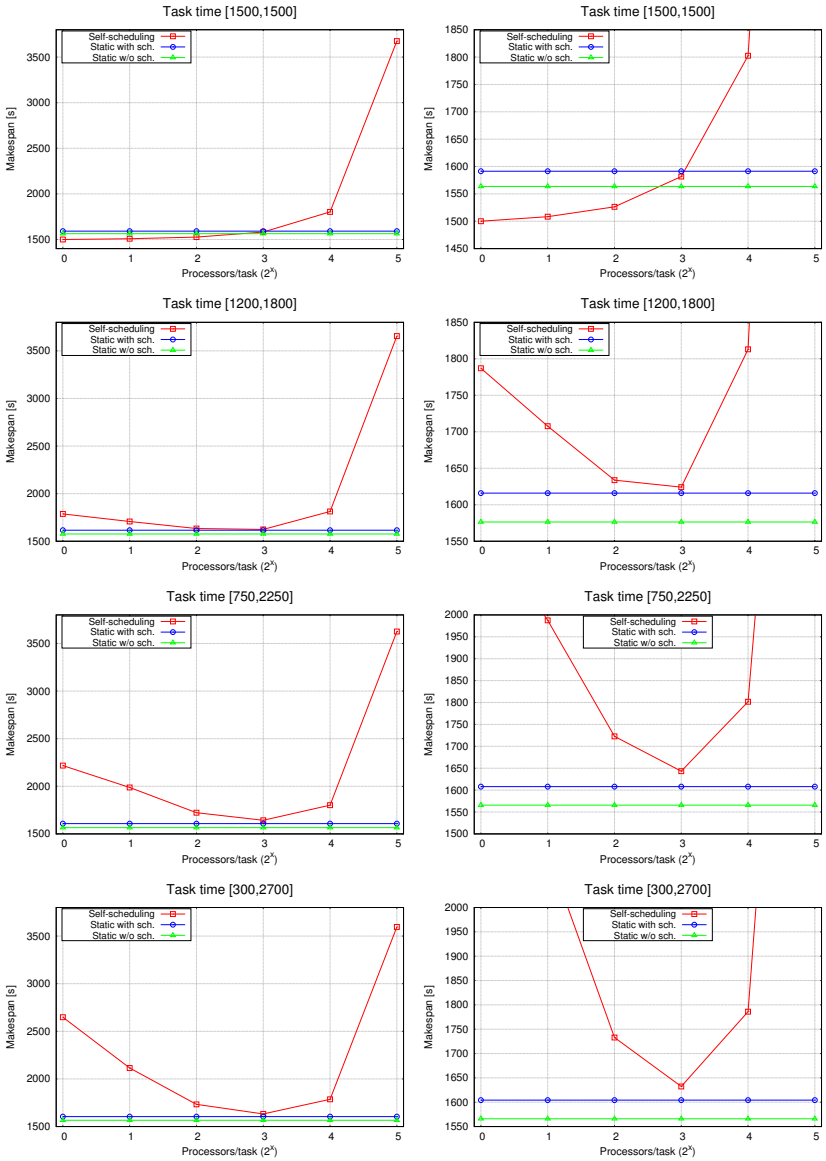


Figure 3.28: Comparison of the average makespan (in seconds) both considering and without considering the scheduling time for batches of 32 long tasks with non-linear scalability obtained by the self-scheduling and static task managers. Same results are represented in the two columns of each row, but a different scale is used.

Self-scheduling with 8 processors/task vs. Static task manager					
Task time	**Self-scheduling makespan [s]**	**Static makespan w/o scheduler [s]**	**Makespan reduction [s]**	**Makespan reduction [%]**	**Computational time reduction in 100 gen. [hours]**
[1200,1800]	1624.15	1576.39	47.76	2.94	42.45
[750,2250]	1643.20	1565.62	77.58	4.72	68.96
[300,2700]	1632.36	1565.96	66.40	4.07	59.02

Self-scheduling with 1 processor/task vs. Static task manager					
Task time	**Self-scheduling makespan [s]**	**Static makespan w/o scheduler [s]**	**Makespan reduction [s]**	**Makespan reduction [%]**	**Computational time reduction in 100 gen. [hours]**
[1200,1800]	1787.05	1576.39	210.66	11.79	187.25
[750,2250]	2217.58	1565.62	651.96	29.40	579.52
[300,2700]	2648.12	1565.96	1082.16	40.87	961.92

Self-scheduling with 16 processors/task vs. Static task manager					
Task time	**Self-scheduling makespan [s]**	**Static makespan w/o scheduler [s]**	**Makespan reduction [s]**	**Makespan reduction [%]**	**Computational time reduction in 100 gen. [hours]**
[1200,1800]	1813	1576.39	236.61	13.05	210.32
[750,2250]	1801.75	1565.62	236.13	13.11	209.89
[300,2700]	1786.03	1565.96	220.07	12.32	195.62

Table 3.1: Makespan reductions achieved by the static task manager compared to different configurations of the self-scheduling algorithm when simulating batches of 32 long tasks with non-linear scalability. The column on the right contains the computational time reductions achieved after running 100 generations.

manager, it can be seen that the latter achieved a reduction of the makespan in the range 10-40%, which involves computational time savings between 180 and 960 hours every 100 generations. The reason is that even if the parallel efficiency of the tasks was maximized, the computational load balance of the batch suffered a degradation caused by the simultaneous execution of many tasks. The second option consists on minimizing the number of tasks being executed simultaneously, i.e. increasing the number of processors assigned per task. However, care must be taken so as to avoid excessive decrease of the parallel efficiency of each task. A common practice is to fix a minimum acceptable value for a task's parallel efficiency, e.g. 80%. In the example shown in Fig. 3.27, it can be calculated that tasks have a parallel efficiency of 83% when executed in 16 processors, what means that 16 would be the maximum number

of processors allowed per task. If the results obtained with this configuration of the self-scheduling task manager are compared with those provided by the static task manager, it can be seen that the latter achieved a reduction of the makespan in the range 12-13%, which involves computational time savings of around 200 hours every 100 generations. Choosing this second option works better in the proposed case study, although the results obtained by the self-scheduling algorithm were outperformed by those of the static task manager.

Having similar information to that shown in Figure 3.28 would have allowed optimizing the configuration of the self-scheduling task manager by assigning 8 processors per task. The static task manager would have achieved a reduction of the makespan in the range 3-5% in this case, which involves computational time savings between 40 and 70 hours every 100 generations. Nevertheless, it is very uncommon that the user has access to such information.

Although the ability of performing accurate task time estimations has been assumed so far, errors will unavoidably arise when handling batches of real tasks. The negative impact of such errors on the results provided by the static task manager is studied in Fig. 3.29. The average makespan of a batch of tasks without considering the scheduling time has been the selected metric in order to quantify the degradation of the load balance. If the results obtained by the static task manager are compared with those provided by the self-scheduling algorithm no matter 1 or 16 processors were assigned per task, it can be seen that the static method outperformed the self-scheduling method even when task time estimation errors with a standard deviation of 10% took place. Moreover, in simulations with task computational times in the ranges [750,2250] and [300,2700], the static task manager obtained better results in spite of time estimation errors with σ_{error}=30% than the self-scheduling algorithm configured for assigning 1 processor per task.

The dynamic task manager is better suited for handling task time estimation errors than the static task manager and for these experiments it has been configured so that the scheduling algorithm is run either twice or three times per batch of tasks. The results provided by the self-scheduling, static and dynamic task managers are overlapped in Figs. 3.30 and 3.31. In the case of the static and dynamic algorithms, the results corresponding to the best obtained makespan including the scheduling time are represented. The results provided by the static and dynamic algorithms have been represented as horizontal lines, even if each line corresponds to a single run of the optimizer (i.e. the evaluation of 10 batches of tasks). The aim is to ease the comparison with the results obtained by the self-scheduling task manager by making visible the

intersection between the lines.

The obtained results show that the dynamic task manager outperformed the static one in every case, proving its superiority. Regarding the optimal number of times the scheduling algorithm should be run, the answer is not clear. The trend observed in Figs. 3.30 and 3.31 is that the higher the heterogeneity in tasks' computational times, the greater the makespan reduction caused by increasing the number of scheduler runs. For tasks' computational times in the range [1200,1800], all depicted results were obtained carrying out 2 scheduler runs per batch. For tasks in the range [750,2250], two results were obtained running 2 schedulers per batch and one running 3 schedulers per batch. Finally, all results were obtained carrying out 3 scheduler runs per batch for tasks' computational times in the range [300,2700]. According to these results, the optimal number of scheduler runs is not related to the quality of the tasks' run time estimations but to the heterogeneity in tasks' computational times, although using the dynamic task manager is more advantageous as the tasks' time estimation error increases.

Makespan reductions achieved by the dynamic task manager when compared to different configurations of the self-scheduling algorithm are shown in Table 3.2. Note that only the best results obtained by the dynamic task manager have been included. Let us compare the average makespan of the batch of tasks taking into account the scheduling overhead. Assuming that the user configured the self-scheduling algorithm as expected, i.e. assigning either 1 or 16 processors per task, the static task manager achieved a makespan reduction in the range 10-40%, which involves computational time savings between 170 and 940 hours every 100 generations. The best results provided by the self-scheduling algorithm were also outperformed by the dynamic task manager. It achieved a makespan reduction of around 2%, which involves computational time savings between 25 and 40 hours every 100 generations.

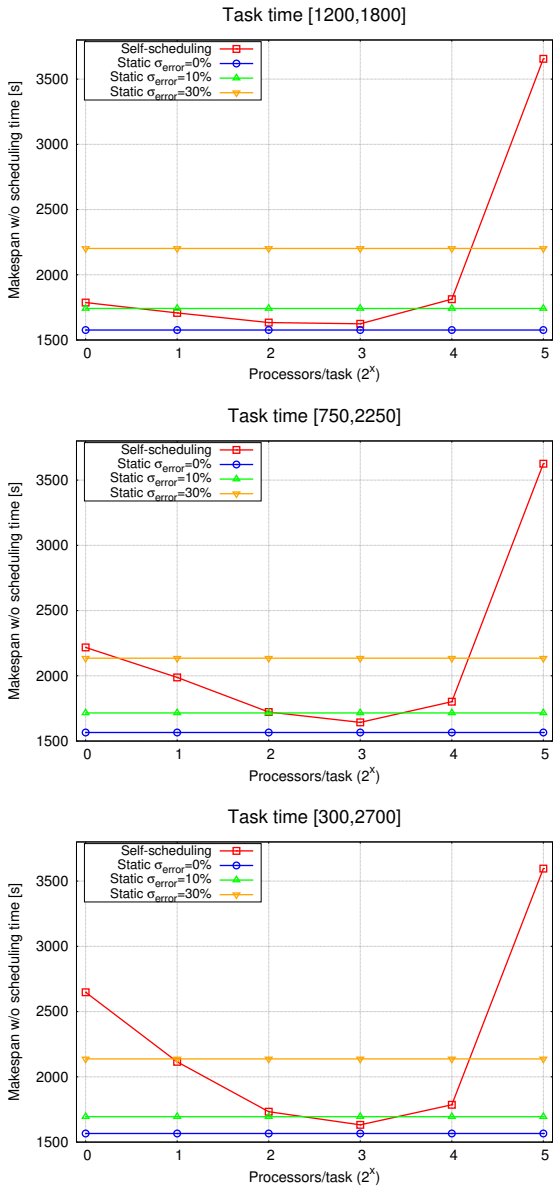


Figure 3.29: Comparison of the average makespan without considering the scheduling time (in seconds) for batches of 32 long tasks with non-linear scalability obtained by the self-scheduling and static task managers and subject to task time estimation errors.

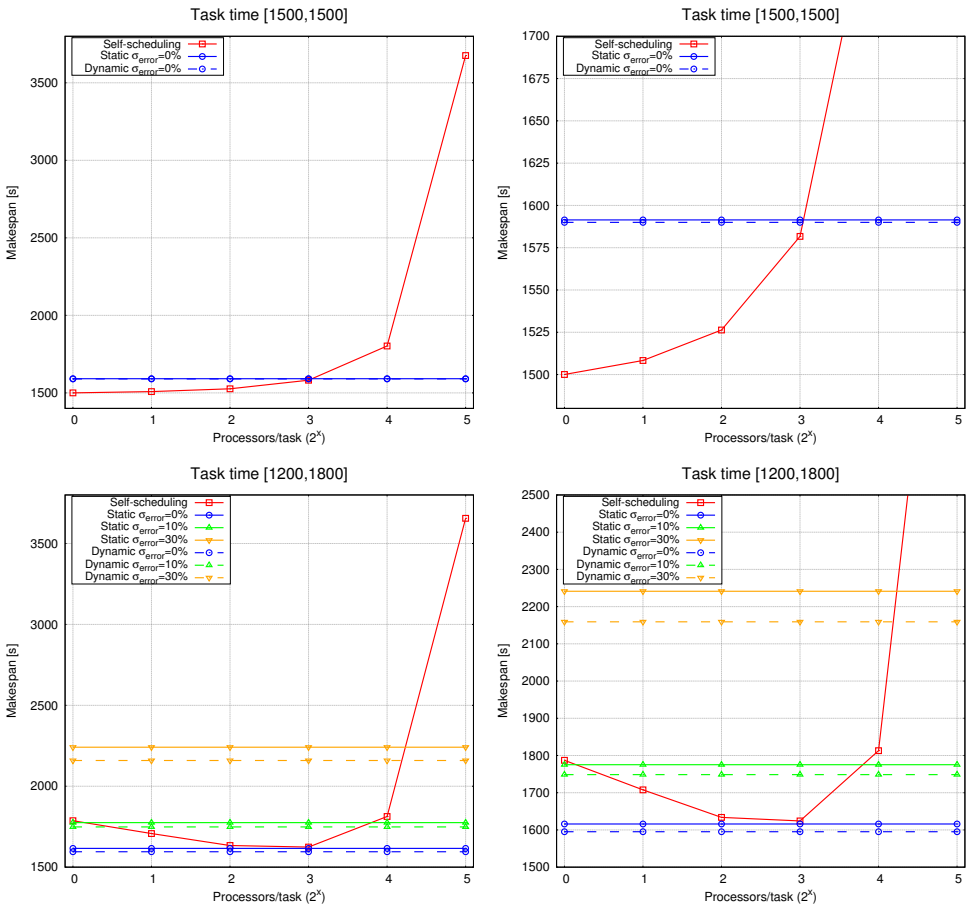


Figure 3.30: Comparison of the average makespan (in seconds) for batches of 32 long tasks with non-linear scalability obtained by the self-scheduling, static and dynamic task managers and subject to task time estimation errors. Results for task times in [750,2250] and [300,2700] may be found in Figure 3.31.

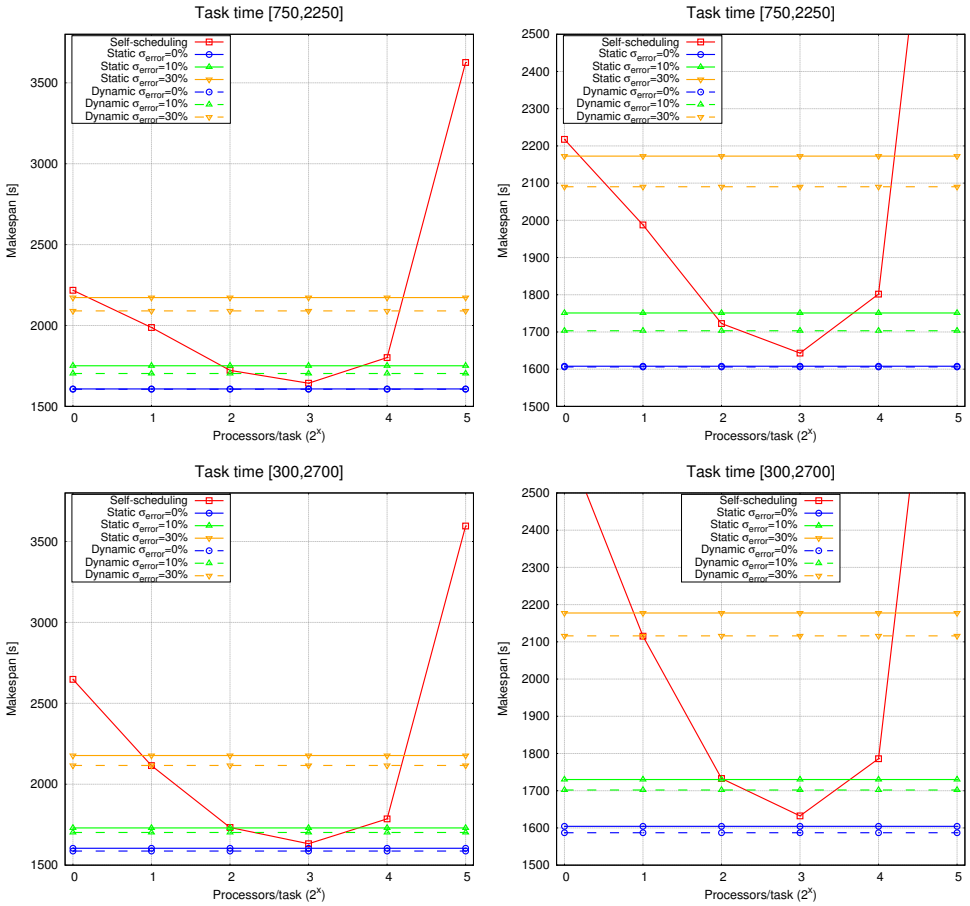


Figure 3.31: Comparison of the average makespan (in seconds) for batches of 32 long tasks with non-linear scalability obtained by the self-scheduling, static and dynamic task managers and subject to task time estimation errors. Results for task times in [1500,1500] and [1200,1800] may be found in Figure 3.30.

Self-scheduling with 8 processors/task vs. Dynamic task manager					
Task time	**Self-scheduling makespan [s]**	**Dynamic makespan [s]**	**Makespan reduction [s]**	**Makespan reduction [%]**	**Computational time reduction in 100 gen. [hours]**
[1200,1800]	1624.15	1595.20	28.95	1.78	25.73
[750,2250]	1643.20	1606.04	37.16	2.26	33.03
[300,2700]	1632.36	1587.13	45.23	2.77	40.20

Self-scheduling with 1 processor/task vs. Dynamic task manager					
Task time	**Self-scheduling makespan [s]**	**Dynamic makespan [s]**	**Makespan reduction [s]**	**Makespan reduction [%]**	**Computational time reduction in 100 gen. [hours]**
[1200,1800]	1787.05	1595.20	191.85	10.74	170.53
[750,2250]	2217.58	1606.04	611.54	27.58	543.59
[300,2700]	2648.12	1587.13	1060.99	40.07	943.10

Self-scheduling with 16 processors/task vs. Dynamic task manager					
Task time	**Self-scheduling makespan [s]**	**Dynamic makespan [s]**	**Makespan reduction [s]**	**Makespan reduction [%]**	**Computational time reduction in 100 gen. [hours]**
[1200,1800]	1813.00	1595.20	217.80	12.01	193.60
[750,2250]	1801.75	1606.04	195.71	10.86	173.96
[300,2700]	1786.03	1587.13	198.90	11.14	176.80

Table 3.2: Makespan reductions achieved by the dynamic task manager compared to different configurations of the self-scheduling algorithm when simulating batches of 32 long tasks with non-linear scalability. The column on the right contains the computational time reductions achieved after running 100 generations.

3.4.5 Concluding remarks

The purpose of this theoretical case study was the performance evaluation of the 3 proposed load balancing strategies, i.e. the self-scheduling, static and dynamic task managers. A first approach was obtained by carrying out simulations of short sleep jobs with linear scalability. The next step was the simulation of long sleep jobs with linear scalability, as their average computational time is more representative of the real target application of the developed load balancing algorithms, i.e. the optimization of CFD & HT models. Finally, long sleep jobs with non-linear scalability were simulated by modeling the degradation of parallel performance taking place as they are assigned an increasing number of processors.

The starting point consisted on proving the existence of the loss of parallel efficiency caused by the computational load imbalance when the self-scheduling task manager is used. This is unavoidable in case of having tasks with heterogeneous computational times, even if the (no. individuals / no. processor groups) ratio is properly selected by the user. Indeed, the main problem when configuring the self-scheduling task manager consists on the impossibility of determining the optimal number of processors that should be assigned per task in order to minimize the makespan of the whole batch of tasks.

The static and dynamic task managers clearly proved their ability to successfully balance the computational load of batches of tasks with heterogeneous completion times, being the main drawbacks the scheduling overhead and handling the task time estimation errors. Some observations can be made regarding the scheduling overhead. On one hand, the longer the scheduling time is, the better task distributions may be obtained. On the other hand, the greater the average completion time of the simulated tasks, the lower the relative scheduling overhead with respect to the average makespan of the batch of tasks. Hence, the static and dynamic task managers maximize their performance the greater the average computational time of the tasks is. Moreover, increasing the convergence speed of the scheduling algorithm will contribute to achieve greater makespan reductions than the ones shown in this theoretical study. Regarding the task time estimation errors, their presence is unfortunately unavoidable and has a negative impact on the load balance achieved by the task managers. According to the obtained results, an acceptable upper bound for estimation errors could be σ_{error}=10%.

The dynamic task manager was the load balancing strategy that achieved the lowest makespans in most cases, being the computational time savings maximized as the heterogeneity in tasks' run time increases. However, the main drawback of the dynamic algorithm consists on the difficulty for determining its optimal configuration in each case.

Note as well that an optimization may comprise the execution of numerous batches of tasks. Consequently, the benefits of reducing the makespan of a batch are multiplied and a significant global impact may be achieved. Moreover, the computational time savings are maximized as an increasing number of processors is used for the optimization. Such a behavior makes the dynamic task manager highly advisable for exascale computing applications.

Finally, it is reminded that the results for each case were obtained by averaging 10 generations. This amount is considered sufficient for this initial study, but should be increased in order to reach more reliable conclusions.

3.5 Implementation and testing of time estimation techniques

After having carried out the state of the art study on time estimation techniques contained in section 3.2.6, it was decided to implement three global data fitting methods in Optimus: kriging interpolation [25], radial basis functions (RBF) [28] and artificial neural networks (ANN) [26]. All of them are available in the Dakota toolkit [23], which was linked to the Optimus library. The user is able to choose among these three methods, as well as a fourth additional option included in Optimus combining kriging interpolation and RBF. In this latter option, compliant with the guidelines provided by Dakota's documentation and configured to be the default option in Optimus, kriging is activated in case the surrogate model is to be built with fewer than 2000 sample points, whereas RBF is preferred in case of having more points.

Some preliminary tests have been performed in order to evaluate the accuracy and computational cost of the aforementioned data fitting methods. Rosenbrock's function (see section 1.8.1), characterized by having a very low evaluation cost and which supports a variable number of dimensions, has been used for this aim. The followed procedure consists of three steps:

1. Two, three and four dimensional Rosenbrock's function has been evaluated several times with the aim of generating samples of size 40, 80, 160, 320, 640 and 1280 points for each case. These samples are later used to feed the surrogate models.

2. With each sample generated in the previous step, 3 surrogate models have been built using the 3 available data fitting methods: kriging interpolation, RBF and ANN. The same processor has always been used so as to be able to compare the computational time required in each case.

3. Finally, Rosenbrock function's values have been estimated using the surrogate models built in step 2. Each test case is characterized by the number of dimensions (2, 3 or 4), the size of the sample and the selected data fitting method, and the obtained surrogate model has been evaluated in 1000 points in order to extract the average estimation error with respect to the values provided by the real Rosenbrock's function.

The obtained results are contained in Table 3.3 and Table 3.4.

Rosenbrock 2 dimensions						
	Computational time [s]			**Average error [%]**		
No. sample points	**Kriging**	**RBF**	**ANN**	**Kriging**	**RBF**	**ANN**
40	0.20	0.22	0.11	8.36	4.39	147.17
80	0.56	0.49	0.14	0.53	0.47	106.07
160	2.86	1.08	0.23	0.38	0.47	44.85
320	13.83	2.73	0.50	0.12	0.07	12.82
640	88.00	3.75	1.54	0.04	0.01	17.94
1280	1114.85	7.41	5.20	0.02	0.01	11.96

Rosenbrock 3 dimensions						
	Computational time [s]			**Average error [%]**		
No. sample points	**Kriging**	**RBF**	**ANN**	**Kriging**	**RBF**	**ANN**
40	0.23	0.23	0.11	9.72	33.96	105.16
80	0.38	0.49	0.14	0.90	3.16	70.19
160	1.03	1.08	0.57	1.92	0.05	19.73
320	4.90	1.96	1.69	0.05	0.06	10.01
640	32.31	3.77	7.25	0.04	0.06	4.68
1280	173.38	7.43	28.36	0.02	0.08	18.64

Rosenbrock 4 dimensions						
	Computational time [s]			**Average error [%]**		
No. sample points	**Kriging**	**RBF**	**ANN**	**Kriging**	**RBF**	**ANN**
40	0.26	0.24	0.12	35.08	143.27	154.91
80	0.25	0.51	0.16	6.52	14.08	47.43
160	0.62	1.10	0.40	1.49	1.44	37.18
320	3.72	1.91	2.41	0.26	1.21	13.19
640	23.52	3.82	16.07	0.08	1.15	4.11
1280	209.49	7.54	105.68	0.05	0.83	4.45

Table 3.3: Preliminary study of the accuracy and computational cost of the data fitting methods implemented in Optimus obtained by evaluating Rosenbrock's function for a variable number of dimensions and sample sizes.

Kriging			
	No. dimensions		
No. sample points	**2**	**3**	**4**
40	0.20	0.23	0.26
80	0.56	0.38	0.25
160	2.86	1.03	0.62
320	13.83	4.90	3.72
640	88.00	32.31	23.52
1280	1114.85	173.38	209.49

RBF			
	No. dimensions		
No. sample points	**2**	**3**	**4**
40	0.22	0.23	0.24
80	0.49	0.49	0.51
160	1.08	1.08	1.10
320	2.73	1.96	1.91
640	3.75	3.77	3.82
1280	7.41	7.43	7.54

ANN			
	No. dimensions		
No. sample points	**2**	**3**	**4**
40	0.11	0.11	0.12
80	0.14	0.14	0.16
160	0.23	0.57	0.40
320	0.51	1.69	2.41
640	1.54	7.25	16.07
1280	5.20	28.36	105.68

Table 3.4: Study of the effect caused by an increasing number of dimensions of the Rosenbrock's function on the computational cost of the data fitting methods implemented in Optimus. The results have been extracted from Table 3.3 and reorganized.

According to Table 3.3, the data fitting method which showed the poorest accuracy was ANN in every case. Kriging interpolation obtained the best results for 3 and 4 dimensional Rosenbrock's functions, whereas it was outperformed by RBF in case of having 2 dimensions. Regarding the computational cost, it can be seen that it increases the larger the amount of sample points used for building the surrogate model, independently of the selected data fitting method. If the three data fitting methods are compared to each other, kriging is usually the one requiring the greatest computational time. This is compliant with the recommendations provided by Dakota's documentation, where the use of this method is discouraged in case of having large samples. For small samples, as well as for every test using the 2-dimensional Rosenbrock's function, ANN was the method involving the lowest computational cost. However, RBF demonstrated to be computationally more efficient as the sample size grows.

The effect caused by an increasing number of dimensions of the Rosenbrock's function on the computational cost of the data fitting methods is studied in Table 3.4 and proves that the three methods behave very dissimilarly. On one hand, kriging interpolation showed great sensitivity, especially as the number of sample points is increased. The trend is that the greater the number of dimensions, the lower the computational cost of building the surrogate model. On the other hand, the sensitivity showed by RBF is negligible. Finally, ANN behaves with the opposite trend as kriging, increasing its computational cost together with the number of dimensions of the Rosenbrock's function.

The performed tests confirm that a method combining both kriging interpolation and RBF seems to be a reasonable initial approach to the task times' estimation problem.

3.6 Illustrative example

New load balancing strategies for population-based optimization algorithms have been proposed in this chapter and a theoretical case study has been performed in order to prove their validity. This section provides an illustrative engineering example consisting on the design of the refrigeration system of a power electronic device. A simple parallelizable thermal model has been created and is wanted to be optimized. Simulations have been run using the task management algorithms proposed in section 3.3.2 and the obtained generations' average makespans have been compared.

The thermal model is composed by a certain amount of pipes (micro-channels) and a fluid circulating through them (see Fig. 3.32). The heat generated by the power electronic device is transferred to the fluid and evacuated, making possible for the device to maintain an acceptable temperature. All pipes have the same diameter, which

conditions the number of pipes that may be placed on the device, and this diameter is the only optimization variable. The objective is to maximize the heat extraction from the power electronic device by deciding if it is better to have a few pipes with a large diameter or many pipes with a small diameter.

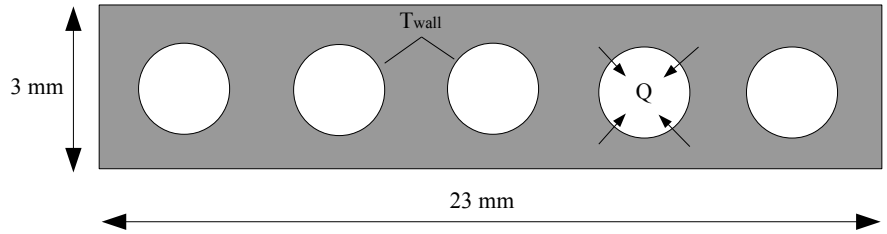


Figure 3.32: Simplified model of a power electronic device (grey rectangle) refrigerated by a fluid circulating through a certain number of micro-channels (white circles).

The simplified steady-state thermal model is based on the following assumptions:

- The heat transfer taking place from the power electronic device to the pipes is modeled by fixing the wall temperature (T_{wall}) of every pipe to a constant value. The same T_{wall} is used for all pipes and does not vary throughout the whole length of the pipes.
- A 1-dimensional convection model is used for calculating the heat transfer from the pipes' walls to the fluid circulating through them, being every pipe discretized using 300 control volumes. All control volumes are initialized to a common temperature and the maximum number of iterations for solving each pipe has been limited to 2000.
- The boundary conditions shared by every pipe are the pressure gradient between the inlet and the outlet, and the inlet temperature of the fluid.
- All pipes are equally long and their walls have the same roughness. In addition, the correlations used for calculating the friction factor and the heat transfer coefficient are the same in all of them.
- The resolution of the pipes is decoupled and no interference exists between them. The overall heat transfer from the power electronic device to the pipes is calculated as the sum of the heats transferred independently to each pipe.

The above simplifications involve that exactly the same pipe will be solved several times in order to obtain the overall heat transfer. The purpose has been the creation of an easily parallelizable thermal model, whose simulation times could also be estimated with a high accuracy. The presented thermal model fulfills both requirements, being the distribution of pipes among different processors easy to handle (each pipe can be assigned to any single processor) and the simulation time dependent only on two variables: the diameter of the pipes and the number of assigned processors. The diameter of the pipes can range in the interval [0.0003, 0.002875] meters and the width of the power electronic device has been fixed to 0.023m. Thus, the minimum number of required micro-channels is 8 and the maximum number 76.

The provided information does not guarantee at all the reproducibility of the simulations that have been carried out. Nevertheless, the aim of this illustrative example is just to emulate the computational load imbalance happening in population-based optimization algorithms by studying a case which involves real data processing. The theoretical case study included in section 3.4, which makes use of sleep jobs and is easy to reproduce, should be enough for the sake of understanding the behavior of the proposed load balancing algorithms.

The parallel performance of the thermal model is a key aspect taken into account by the task management algorithms. Moreover, the scalability of the model varies depending on the number of pipes that is utilized. Some speedup tests have been carried out with models composed by 8, 42 and 76 pipes so as to have a graphical representation of this variation. The study was carried out assigning 1, 2, 4, 8, 16 and 32 processors for the resolution of each model and results obtained from 16 simulations were averaged for each number of processors. This accounts for a total amount of 96 simulations in order to obtain the speedup graph of each one of the 3 cases, which are shown in Fig. 3.33.

It can be seen that the parallel speedup of the model is close to linear up to 8 processors. From this point on, the degradation of parallel efficiency begins to be noticeable in every case. The most remarkable behavior is that of the model composed by 8 pipes, for which it is impossible to increase the speedup when more than 8 processors are assigned. The reason is that a single pipe cannot be simulated in parallel and this model has no more than 8 pipes. Even if additional processors are assigned to it, there is no workload that can be placed in those processors and they remain idle.

No real optimization is carried out in this illustrative example, being the reasons twofold. First, because the only relevant result is the generations' average makespan and not obtaining the optimal pipe diameter. Second, because the heterogeneity of the individuals in the population is wanted to be preserved and optimization processes tend

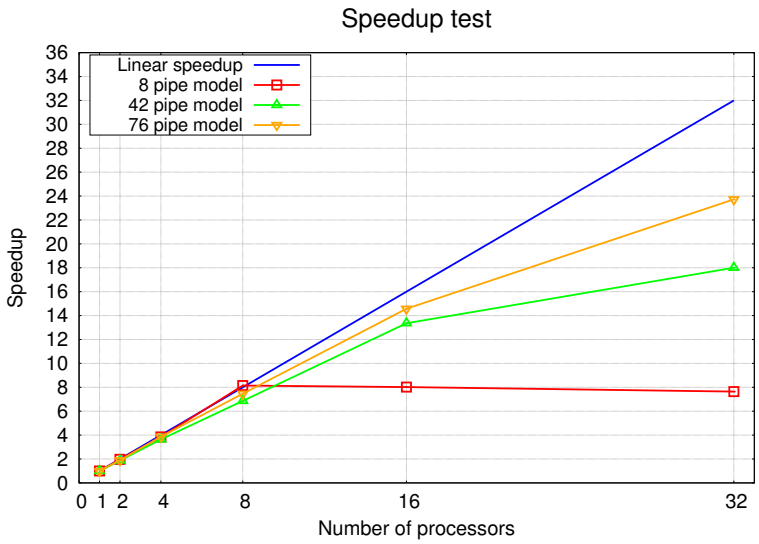


Figure 3.33: Comparison of the parallel speedup of the simplified thermal model in case it is composed by 8, 42 and 76 pipes.

to make populations more homogeneous. It is reminded that the higher the homogeneity level, the easier to balance the computational load of the generations. Therefore, the evolution engine of the optimization algorithm has been deactivated and the individuals contained in every generation have been created randomly according to a uniform distribution of the diameter values in the range [0.0003, 0.002875] meters.

Using the load balancing strategies proposed in Chapter 3 requires the availability of a database containing information of past evaluations of individuals so that the new methods are able to estimate as accurately as possible the evaluation times of the new individuals to be processed. Hence, several sampling simulations were executed so as to create the aforementioned database. In a first step, a sample composed by 320 individuals was created and evaluated. The pipes' diameters and the number of processors assigned to each individual were uniformly generated in [0.0003, 0.002875] and $\{1,2,4,8,16,32\}$ respectively. In a second step, and with the aim of ensuring the availability of a minimum amount of information for every number of processors, 6 additional generations composed by 32 individuals uniformly created in the interval [0.0003, 0.002875] were evaluated. However, the number of processors assigned per individual was not random and was progressively increased: 1 processor/individual in the first

generation, 2 processors/individual in the second generation, 4 processors/individual in the fourth generation, etc. These 6 generations gave rise to a total amount of 192 new samples.

At this point, a particularity of the present optimization problem must be noted. Being the optimization variable the diameter of the pipes, the number of pipes that can be installed in the refrigeration device is represented in Fig. 3.34. It can be seen that the smaller the diameter of the pipes, the greater the sensitivity of the number of pipes to diameter variations. The consequence is that a uniform sample in the interval of allowed diameters does not involve having uniform data regarding the number of pipes. Since the computational cost of solving the thermal model depends on the latter, it is crucial to sample this variable appropriately. Thus, 3 additional sampling simulations were carried out using diameters belonging to 3 sub-intervals contained in the original bounds: i) [0.0003, 0.000418] meters, leading to [55, 76] pipes; ii) [0.000418, 0.00076] meters, leading to [30, 55] pipes; and iii) [0.00076, 0.002875] meters, leading to [8, 30] pipes. Each sampling simulation gave rise to a total amount of 192 new samples and was composed by 6 generations of 32 individuals, being diameter values uniformly created in the previously mentioned intervals. The number of processors assigned per individual was not random and was progressively increased: 1 processor/individual in the first generation, 2 processors/individual in the second generation, 4 processors/individual in the fourth generation, etc.

The execution of all the aforementioned sampling simulations generated a total amount of 1088 sample points. Note also that further development of time estimation techniques is required in order to optimize the design of sampling simulations.

After having generated the necessary samples, an appropriate configuration for the optimization algorithm was decided prior to testing the behavior of the available task managers. The genetic algorithm implemented in Optimus was parametrized in a similar way to that used in the theoretical case study in section 3.4, utilizing the peer partition approach (see section 1.7.2) and fixing a population size of 32 individuals to be run in 32 processors. The individuals are randomly created in each generation, being the diameter of the pipes defined according to a uniform probability distribution in the interval [0.0003, 0.002875] meters in order to preserve the population's heterogeneity regarding the computational load.

The computational time required for solving an individual depends on the number of pipes in the thermal model. 3 preliminary tests have been performed with the aim of obtaining some references of this cost and the obtained results are shown in Table 3.5. Every individual was run in 1 processor and the computational time needed for a given

number of pipes was calculated by averaging the behavior of 16 individuals. Note that a standard deviation of around 2.5% was always obtained in spite of using the same processors node for every simulation. This fact might be suggesting the impossibility of estimating evaluation times of individuals with a higher accuracy in cases involving real data processing.

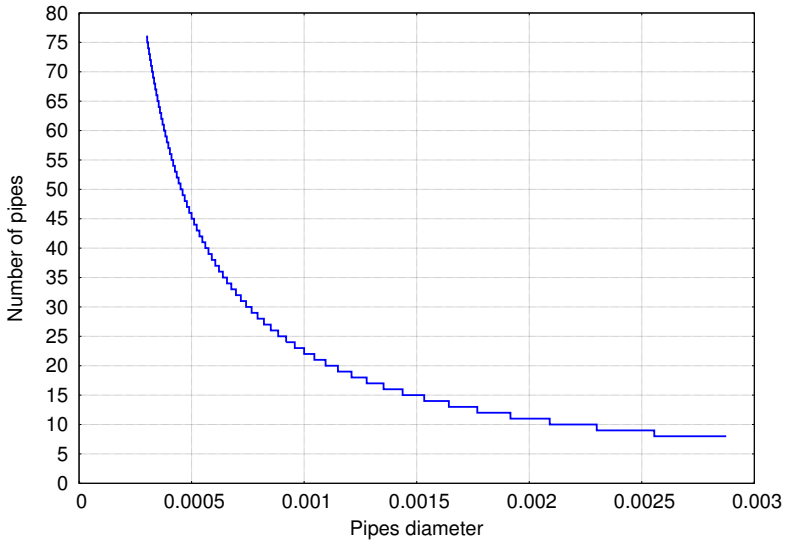


Figure 3.34: Graph showing the number of pipes that can be installed in the refrigeration device for the whole range of diameter values.

No. pipes	Avg. comp. time [s]	Avg. comp. time [min]	Std. Dev. [%]
8	325	5.4	2.40%
42	1837	30.0	2.50%
76	4727	78.8	2.70%

Table 3.5: Results of the preliminary tests for measuring the computational time required for solving the thermal model when it is composed by a variable number of pipes.

A case study has been designed in order to compare the performance of different configurations of the 3 available task managers:

- Self-scheduling task manager: 6 simulations were carried out assigning 1, 2, 4, 8, 16 and 32 processors per individual.
- Static task manager: 3 simulations were carried out, being the time limitation for the scheduling algorithm equal to 1%, 2% and 3% of the previous generation's makespan. The task manager was able to assign 2^x processors per individual, being x in the range [0,5]. The evaluation time of new individuals was estimated by means of kriging interpolation (see section 3.2.6).
- Dynamic task manager: A total amount of 6 simulations was carried out, being the scheduling algorithm run twice in half of them and 3 times in the other half of simulations. In the 3 simulations contained in each half, the time limitation for the scheduling algorithm was equal to 1%, 2% and 3% of the previous generation's makespan. The task manager was able to assign 2^x processors per individual, being x in the range [0,5]. The evaluation time of new individuals was estimated by means of kriging interpolation.

Each simulation was composed of 10 generations, which have been averaged to obtain the makespans achieved by the different task managers. Note that even if the same database of sample points was used in the first generation of every simulation carried out by the static and dynamic load balancing algorithms, additional information regarding the evaluation time of new individuals is stored in run time. Hence, the surrogate models used in subsequent generations of every simulation differ from each other.

The results provided by the simulations run using the static and dynamic task managers are shown in Fig. 3.35. On one hand, the static task manager was tested in cases 1-3, being the scheduling time limited to 1% (case 1), 2% (case 2) and 3% (case 3). On the other hand, the dynamic task manager was tested in cases 4-9. The scheduling algorithm was run twice per generation in cases 4-6 and 3 times per generation in cases 7-9, being the scheduling time limited to 1% in cases 4 and 7, to 2% in cases 5 and 8, and to 3% in cases 6 and 9.

The average generations' makespan for all tested configurations is depicted in the upper graph of Fig. 3.35. In case of using the static task manager (cases 1-3), it can be seen that the makespan increases as the time limitation for the scheduling algorithm is increased. Consequently, the lowest makespan was achieved for a scheduling time

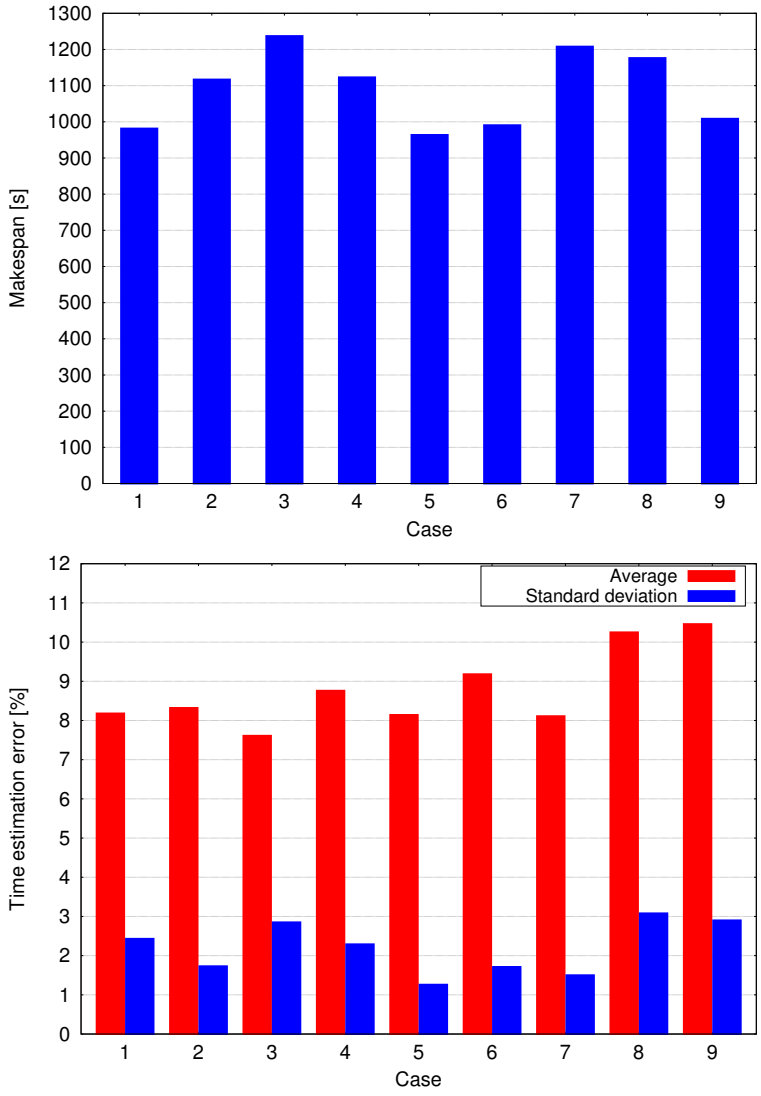


Figure 3.35: Average generations' makespans and time estimation errors obtained in cases 1-9.

limitation of 1% (case 1). In case of using the dynamic task manager (cases 4-9), the makespans obtained when the scheduling algorithm was called 3 times per generation are greater than those obtained when it was called only twice. For the latter configuration the lowest makespan was achieved in case 5 (scheduling time limitation of 2%), whereas case 9 (scheduling time limitation of 3%) achieved the best results when the scheduling algorithm was run 3 times per generation.

Nevertheless, different values for the average and standard deviation of the time estimation errors were obtained in the performed simulations, as it is shown in the lower graph of Fig. 3.35. Consequently, it is difficult to carry out a fair comparison of the different configurations utilized for the static and dynamic task managers. Making the time estimation errors more homogeneous would require either increasing the number of sample points in order to increase the predictions' accuracy, or calculating averages based on more than the currently used 10 generations. But since the aim of this illustrative example consists on proving that the new load balancing methods are able to outperform the self-scheduling task manager when individuals involving real data processing are evaluated, the quality of the obtained results is considered satisfactory. The reader is referred to the theoretical case study carried out in section 3.4 in order to compare different configurations of the static and dynamic task managers.

Taking the previous considerations into account, the best results provided by the static task manager (case 1) and the dynamic task manager running the scheduling algorithm twice (case 5) and 3 times (case 9) have been compared with the results obtained by the self-scheduling task manager in Fig. 3.36. It can be seen that the 3 cases in which the new load balancing algorithms were used outperformed the best makespan obtained by the self-scheduling task manager. This fact is very meaningful of the superior capacity of the new task managers for optimizing the computational load distribution, and even more if it is taken into account that the average time estimation errors were above 7% in every case.

The best average makespan (963.8 seconds) was achieved by the dynamic task manager running the scheduling algorithm twice per generation and with a scheduling time limitation of 2% (case 5). The self-scheduling task manager was unable to obtain a makespan lower than 1079.44 seconds, what involves that a time saving of 115.64 seconds (10.71%) was achieved by the dynamic task manager. Further information of the makespan reductions obtained in case 5 in comparison to different configurations of the self-scheduling algorithm is shown in Table 3.6).

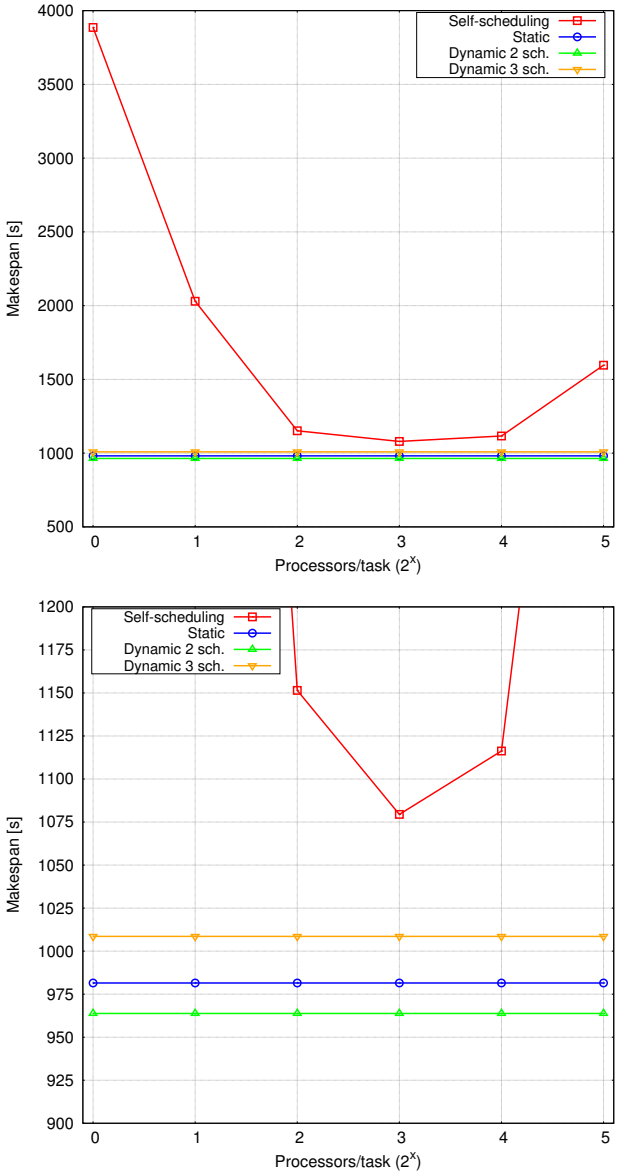


Figure 3.36: Comparison of the average generations' makespans (in seconds) obtained by the self-scheduling task manager and the new load balancing strategies. Same results are represented in both graphs, but a different scale is used.

Self-scheduling vs. Best result (case 5)					
Self-scheduling processors/task	**Self-scheduling makespan [s]**	**Case 5 makespan [s]**	**Makespan reduction [s]**	**Makespan reduction [%]**	**Comp. time reduction in 100 gen. [h]**
1	3885.92	963.80	2922.12	75.20	2597.44
8	1079.44	963.80	115.64	10.71	102.79
16	1116.27	963.80	152.47	13.66	135.53

Table 3.6: Makespan reductions achieved in case 5 compared to different configurations of the self-scheduling algorithm. The column on the right contains the computational time reductions achieved after running 100 generations.

If it is taken into account that the parallel speedup of the thermal model (see Fig. 3.33) is the only information available when configuring the self-scheduling task manager, the user can decide between two options.

The first one consists on maximizing the parallel efficiency of individuals (i.e. reducing the number of processors assigned per individual) in the hope that this strategy will lead to minimize the generations' makespans. However, if the results obtained by the self-scheduling task manager with a single processor assigned per individual are compared to those obtained in case 5, it can be seen that the latter achieved a reduction of the average makespan of 75%, which involves computational time saving of around 2600 hours every 100 generations. The reason is that even if the efficiency of the individuals was maximized, the computational load balance of the entire generation suffered a degradation caused by the simultaneous execution of many individuals.

The second option consists on minimizing the number of individuals being executed simultaneously, i.e. increasing the number of processors assigned per individual. However, care must be taken so as to avoid excessive decrease of the parallel efficiency of each individual. A common practice is to fix a minimum acceptable value for an individual's parallel efficiency, e.g. 80%. In this particular example, the parallel performance graph varies depending on the number of pipes of the thermal model. The maximum number of pipes is 76 and the minimum number 8, having the model 42 pipes in average. In this latter case, the parallel efficiency of the model if 16 processors are assigned is of 83%, being consequently 16 the maximum number of processors allowed per individual. If the results obtained with this configuration of the self-scheduling task manager are compared with those provided by case 5, it can be seen that the latter achieved a reduction of the makespan of 13.66%, which involves computational time savings of around 135 hours every 100 generations.

The obtained results show the difficulty of balancing the computational load of even the simplest engineering model during its optimization, but the strategies developed in the scope of this Doctoral Thesis proved to be a potential solution to this problem.

3.7 Conclusions

Load balancing methods for parallel optimization algorithms have been studied and developed in this third chapter. The approach to the load balancing problem was started by introducing the divisible load theory and the concept of job flexibility. After that, the main factors affecting the parallel performance of optimization algorithms were identified and divided into three groups: factors related to the hardware, factors related to the nature of tasks and factors related to the scheduler's configuration.

The next step consisted on defining the methodology for developing and testing new load balancing algorithms, and it was decided to split the load balancing problem into 2 subproblems: a tasks' time estimation problem and a task scheduling combinatorial problem. Regarding the task scheduling problem, a state of the art study was carried out including the review of classical combinatorial problems and of other applications using similar scheduling algorithms, such as the Resource and Job Management Systems (RJMS) installed in High Performance Computing (HPC) platforms. A second state of the art study was carried out in order to tackle the subproblem consisting on estimating the tasks' computational time. Several data fitting methods were analyzed, as well as some suggestions made by other authors.

Then, a task scheduling algorithm was designed and included in two different task managers: the static task manager and the dynamic task manager. At this point, 3 task managers (self-scheduling, static and dynamic) were available but there was no information regarding their performance. Therefore, some theoretical case studies were carried out, which consisted in simulating short cases with linear scalability, long cases with linear scalability and long cases with non-linear scalability. The tests confirmed the superiority of the dynamic task manager subject to an appropriate configuration and to the availability of accurate enough tasks' time estimations.

Although the tasks' time estimation subproblem was not properly tackled due to lack of time, 3 global data fitting methods were implemented for this aim: kriging interpolation, radial basis functions (RBF) and artificial neural networks (ANN). Moreover, some preliminary tests were performed using the Rosenbrock's function in order to evaluate their accuracy and computational cost.

Finally, an illustrative example consisting on the optimization of a power electronic device was presented in order to extrapolate the conclusions obtained in the theoretical case study to engineering cases involving real data processing. Despite the apparent simplicity of the studied thermal model, the load balancing strategies developed in this chapter outperformed the results obtained by the self-scheduling task manager.

Acknowledgments

This work has been financially supported by a FPU Doctoral Grant (Formación de Profesorado Universitario) awarded by the Ministerio de Educación, Cultura y Deporte, Spain (FPU12/06265) and by Termo Fluids S.L. The author thankfully acknowledges these institutions.

References

[1] S. Gepner, J. Majewski, and J. Rokicki. Dynamic load balancing for adaptive parallel flow problems. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, editors, *Parallel processing and applied mathematics: 8th International Conference, PPAM 2009, Wroclaw, Poland, September 13-16, 2009. Revised Selected Papers, Part I*, pages 61–69, Berlin, Heidelberg, 2010. Springer.

[2] Z. H. Zhan, G. Y. Zhang, Y. J. Gong, and J. Zhang. Load balance aware genetic algorithm for task scheduling in cloud computing. In *Asia-Pacific Conference on Simulated Evolution and Learning*, pages 644–655. Springer International Publishing, 2014.

[3] V. Bharadwaj, D. Ghose, and T. G. Robertazzi. Divisible load theory: A new paradigm for load scheduling in distributed systems. *Cluster Computing*, 6(1):7–17, 2003.

[4] D. G. Feitelson and L. Rudolph. Toward convergence in job schedulers for parallel supercomputers. In *Job scheduling strategies for parallel processing: IPPS '96 Workshop Honolulu, Hawaii, April 16, 1996*, pages 1–26, Berlin, Heidelberg, 1996. Springer.

[5] G. Utrera, J. Corbalan, and J. Labarta. Scheduling parallel jobs on multicore clusters using cpu oversubscription. *The Journal of Supercomputing*, 68(3):1113–1140, 2014.

[6] Y. Georgiou. *Contributions for resource and job management in high performance computing*. PhD thesis, Université de Grenoble, France, 2010.

[7] D. M. Valdivieso. *Towards a virtual platform for aerodynamic design, performance assessment and optimization of Horizontal Axis Wind Turbines*. PhD thesis, Universitat Politècnica de Catalunya, Spain, 2017.

[8] P. Nordin, R. Braun, and P. Krus. Job-scheduling of distributed simulation-based optimization with support for multi-level parallelism. In *Proceedings of the 56th Conference on Simulation and Modelling (SIMS 56), October, 7-9, 2015, Linköping University, Sweden*, number 119, pages 187–197. Linköping University Electronic Press, 2015.

[9] B. Hamidzadeh, L. Y. Kit, and D. J. Lilja. Dynamic task scheduling using online optimization. *IEEE Trans. Parallel Distrib. Syst.*, 11(11):1151–1163, November 2000.

[10] Y. Yang, K. van der Raadt, and H. Casanova. Multiround algorithms for scheduling divisible loads. *IEEE Transactions on Parallel and Distributed Systems*, 16(11):1092–1102, Nov 2005.

[11] T. Blochwitz, M. Otter, J. Akesson, M. Arnold, C. Clauss, H. Elmqvist, M. Friedrich, A. Junghanns, J. Mauss, D. Neumerkel, et al. Functional mockup interface 2.0: The standard for tool independent exchange of simulation models. In *Proceedings of the 9th International MODELICA Conference; September 3-5; 2012; Munich; Germany*, number 076, pages 173–184. Linköping University Electronic Press, 2012.

[12] W. Zhou and Y. P. Bu. An adaptive genetic algorithm for the grid scheduling problem. In *2012 24th Chinese Control and Decision Conference (CCDC)*, pages 730–734. IEEE, 2012.

[13] E. Y. H Lin. A bibliographical survey on some well-known non-standard knapsack problems. *INFOR: Information Systems and Operational Research*, 36(4):274–317, 1998.

[14] Jr. E. G. Coffman, M. R. Garey, and D. S. Johnson. Approximation algorithms for NP-hard problems. pages 46–93. PWS Publishing Co., Boston, MA, USA, 1997.

[15] A. Lodi, S. Martello, and D. Vigo. Recent advances on two-dimensional bin packing problems. *Discrete Appl. Math.*, 123(1-3):379–396, November 2002.

[16] H. Izakian, A. Abraham, and V. Snasel. Comparison of heuristics for scheduling independent tasks on heterogeneous distributed environments. In *Proceedings of the 2009 International Joint Conference on Computational Sciences and Optimization - Volume 01*, CSO '09, pages 8–12, Washington, DC, USA, 2009. IEEE Computer Society.

[17] A. Ecer, Y. P. Chien, H. U. Akay, and J. D. Chen. Load balancing for multiple parallel jobs. In *European Congress on Computational Methods in Applied Sciences and Engineering ECCOMAS; September 11-14; 2000*, Barcelona, Spain, 2000.

[18] F. A. Omara and M. M. Arafa. Genetic algorithms for task scheduling problem. *Journal of Parallel and Distributed Computing*, 70(1):13–22, 2010.

[19] M. Tripathy and C. R. Tripathy. Dynamic load balancing with "work stealing" for distributed shared memory clusters. In *2010 International Conference on Industrial Electronics, Control & Robotics (IECR)*, pages 43–47. IEEE, 2010.

[20] M. M. Najafabadi, M. Zali, S. Taheri, and F. Taghiyareh. Static task scheduling using genetic algorithm and reinforcement learning. In *2007 IEEE Symposium on Computational Intelligence in Scheduling*, pages 226–230. IEEE, 2007.

[21] C. Franke, J. Lepping, and U. Schwiegelshohn. Greedy scheduling with complex obejectives. In *2007 IEEE Symposium on Computational Intelligence in Scheduling*, pages 113–120. IEEE, 2007.

[22] A. Nissimov and D. G. Feitelson. Probabilistic backfilling. In E. Frachtenberg and U. Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing: 13th International Workshop, JSSPP 2007, Seattle, WA, USA, June 17, 2007. Revised Papers*, pages 102–115, Berlin, Heidelberg, 2008. Springer.

[23] B. M. Adams, L. E. Bauman, W. J. Bohnhoff, K. R. Dalbey, M. S. Ebeida, J. P. Eddy, M. S. Eldred, P. D. Hough, K. T. Hu, J. D. Jakeman, J. A. Stephens, L. P. Swiler, D. M. Vigil, and T. M. Wildey. Dakota, a multilevel parallel object-oriented framework for design optimization, parameter estimation, uncertainty quantification, and sensitivity analysis: version 6.0 user's manual. Sandia Technical Report SAND2014-4633, July 2014. Updated November 2015 (Version 6.3).

[24] R. H. Myers, D. C. Montgomery, and C. M. Anderson-Cook. Response surface methodology: product and process optimization using designed experiments, 2009.

[25] A. A. Giunta and L. T. Watson. A comparison of approximation modeling techniques: Polynomial versus interpolating models. In *7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, number AIAA-98-4758, St. Louis, MO, 1998*, pages 392–404. American Institute of Aeronautics and Astronautics, 1998.

[26] D. C. Zimmerman. Genetic algorithms for navigating expensive and complex design spaces. *Final Report for Sandia National Laboratories contract AO-7736 CA*, 2:30332–0150, 1996.

[27] J. H. Friedman. Multivariate adaptive regression splines. *The Annals of Statistics*, 19(1):1–141, 1991.

[28] M. J. L. Orr. Introduction to radial basis function networks, 1996.

[29] A. Nealen. An as-short-as-possible introduction to the least squares, weighted least squares and moving least squares methods for scattered data approximation and interpolation. *Technical report, Discrete Geometric Modeling Group, Technishe Universitaet, Berlin, Germany, 2004, URL: http://www.nealen.com/projects*.

[30] V. E. Bazterra, M. Cuma, M. B. Ferraro, and J. C. Facelli. A general framework to understand parallel performance in heterogeneous clusters: analysis of a new adaptive parallel genetic algorithm. *Journal of Parallel and Distributed Computing*, 65(1):48–57, 2005.

4

Conclusions and future work

4.1 Conclusions

The objective of this Doctoral Thesis was the development of parallel optimization algorithms to be deployed in massively parallel computers. A generic mathematical optimization tool applicable in any field of science and engineering has been developed for this aim, although a special focus has been put on the application of the library to the fields of expertise of the institution hosting this research activity, i.e. the Heat and Mass Transfer Technological Center (CTTC). The main contribution has been the development and implementation of several load balancing strategies which have demonstrated to be able to reduce the simulation time required by the optimization tool. This was proved by an exhaustive theoretical case study. Finally, the time reduction attained in a real-world engineering application that consists on optimizing the refrigeration system of a power electronic device was presented as an illustrative example. A summary of the conclusions extracted in the previous chapters is provided in the subsequent paragraphs.

A thorough state of the art study was conducted in the first chapter with the aim of obtaining a global scope of the mathematical optimization techniques available to date. Special emphasis was laid on the genetic algorithm. Then a description of the main concepts and shortcomings of the standard parallelization strategies available for such population-based optimization methods was included.

Since the computational cost of a real-world design problem may be considerable, the development of appropriate parallelization techniques is a critical issue in order to benefit from optimization strategies. The goal of every parallelization strategy is to maximize the CPU usage by minimizing the communication overhead and by balancing the computational load correctly, thus avoiding idleness of allocated processors. In the case of traditional optimization algorithms, and particularly of genetic algorithms, the principal causes of computational load imbalance are the following:

- Inappropriate ratio (no. individuals / no. processor groups): Usually, when several individuals may be evaluated simultaneously, each individual is assigned to a group of processors. In standard algorithms, all groups are formed by the same number of processors and remain unchanged during optimization. However, if the ratio (no. individuals / no. processor groups) is not well set, some processors may remain idle while others are immersed in computation.

- Heterogeneous parallel computer systems: Hardware heterogeneity translates into non-homogeneous objective evaluation times and the consequent loss of parallel performance of optimization algorithms.

- Heterogeneous objective function evaluation time: Heterogeneity in the computational cost of evaluating the objective functions may cause an important load imbalance provided that the simulation time of the genetic algorithm is dominated by the evaluation time of the objective functions. This phenomenon happens when the evaluation cost of individuals is dependent on the optimization variables and is not unusual in heat transfer and nonlinear mechanics applications.

The new optimization library named Optimus was implemented in the second chapter. After carrying out a state of the art study on available optimization libraries, it was decided to base the development of Optimus in two open-source packages: Paradiseo and Trilinos/Moocho. Paradiseo was chosen because of its implementation of genetic algorithms, whereas the availability of gradient-based local search methods was the appealing feature of Trilinos/Moocho. Optimus makes use of the best performing characteristics of both libraries, but additional functionality was also added. Finally, validation tests of the new library were carried out at the end of the chapter. They include the optimization of benchmark mathematical functions and two specific tests from the field of Computational Fluid Dynamics and Heat Transfer (CFD & HT), namely the optimization of the energy efficiency of a fridge and the optimization of the geometry of a pipe. Optimus succeeded in every test, proving the suitability of the library for solving real-world optimization problems.

In the third chapter load balancing methods for parallel optimization algorithms were developed and studied. The approach to the load balancing problem was started by introducing the divisible load theory and the concept of job flexibility. After that, the main factors affecting the parallel performance of optimization algorithms were identified and divided into three groups: factors related to the hardware, factors related to the nature of tasks and factors related to the scheduler's configuration.

The next step consisted on defining the methodology for developing and testing new load balancing algorithms, and it was decided to split the load balancing problem into 2 subproblems: a tasks' time estimation problem and a task scheduling combinatorial problem. Regarding the task scheduling problem, a state of the art study was carried out including the review of classical combinatorial problems and of other applications using similar scheduling algorithms, such as the Resource and Job Management Systems (RJMS) installed in High Performance Computing (HPC) platforms. A second state of

the art study was carried out in order to tackle the subproblem consisting on estimating the tasks' evaluation time. Several data fitting methods were analyzed, as well as some suggestions made by other authors.

Two different task managers were then proposed, namely the static task manager and the dynamic task manager, and three theoretical case studies were carried out in order to evaluate their parallel performance with respect to the traditional self-scheduling task manager. A first approach was obtained by carrying out simulations of short sleep jobs with linear scalability. The next step was the simulation of long sleep jobs with linear scalability, as their average computational time is more representative of the real target application of the developed load balancing algorithms, i.e. the optimization of CFD & HT models. Finally, long sleep jobs with non-linear scalability were simulated by modeling the degradation of parallel performance taking place as they are assigned an increasing number of processors.

The starting point of the tests consisted on proving the existence of the loss of parallel efficiency caused by the computational load imbalance when the self-scheduling task manager is used. This is unavoidable in case of having tasks with heterogeneous computational times, even if the (no. individuals / no. processor groups) ratio is properly selected by the user. Indeed, the main problem when configuring the self-scheduling task manager consists on the impossibility of determining the optimal number of processors that should be assigned per task in order to minimize the makespan of the whole batch of tasks. The static and dynamic task managers clearly proved their ability to successfully balance the computational load of batches of tasks with heterogeneous completion times, being the main drawbacks the scheduling overhead and handling the task time estimation errors. Moreover, the tests confirmed the superiority of the dynamic task manager subject to an appropriate configuration and to the availability of accurate enough tasks' time estimations.

Some observations can be made regarding the scheduling overhead. On one hand, the longer the scheduling time is, the better task distributions may be obtained. On the other hand, the greater the average completion time of the simulated tasks, the lower the relative scheduling overhead with respect to the average makespan of the batch of tasks. Hence, the static and dynamic task managers maximize their performance the greater the average computational time of the tasks is. Regarding the task time estimation errors, their presence is unfortunately unavoidable and has a negative impact on the load balance achieved by the task managers. An acceptable upper bound for estimation errors was provided according to the obtained results.

Note as well that an optimization may comprise the execution of numerous batches of tasks. Consequently, the benefits of reducing the makespan of a batch are multiplied and a significant global impact may be achieved. Moreover, the computational time savings are maximized as an increasing number of processors is used for the optimization. Such a behavior makes the dynamic task manager highly advisable for exascale computing applications.

Although the tasks' time estimation subproblem was not properly tackled due to lack of time, 3 global data fitting methods were implemented for this aim: kriging interpolation, radial basis functions (RBF) and artificial neural networks (ANN). Moreover, some preliminary tests were performed using the Rosenbrock's function in order to evaluate their accuracy and computational cost.

Finally, an illustrative example consisting on the optimization of a power electronic device was presented in order to extrapolate the conclusions obtained in the theoretical case study to engineering cases involving real data processing. Despite the apparent simplicity of the studied thermal model, the load balancing strategies developed in this chapter outperformed the results obtained by the self-scheduling task manager.

4.2 Future work

The inherent parallel nature of evolutionary computation is a promising factor in developing robust and scalable optimization algorithms which can help reach the goal of a successful integration of Optimization and High Performance Computing capabilities. Therefore, parallel evolutionary computation is expected to be an active research area in the near future. The contribution of this Doctoral Thesis has consisted on the proposal of some load balancing algorithms that can be integrated in evolutionary computation software packages to improve their parallel efficiency. Some suggestions to further develop the proposed algorithms are included hereafter.

The tasks' time estimation subproblem is a key aspect of the load balancing algorithms, but it was not studied in detail due to lack of time. Thus, it is crucial to carry out research in this direction so that the proposed load balancing algorithms can be comfortably used for solving production optimization problems. The research should include two aspects: i) selection or development of appropriate data fitting techniques, and ii) definition of a robust sampling method. In this sense, some criterion able to measure the accuracy of the subsequently created tasks' time maps and to activate the load balancing algorithms provided that a certain accuracy level has been reached is to be implemented. The author's experience shows that it is preferable to use the

self-scheduling task manager unless a minimal accuracy level has been reached. The reason behind is that too great time estimation errors are prone to create severe load imbalance, causing the inefficient use of computational resources and serious simulation delays. Note that the parallel execution of data fitting methods may be necessary as the number of points used for modeling time maps increases.

Regarding the task scheduling combinatorial subproblem solved by both the static and dynamic task managers, a rigorous study of the task scheduling algorithm is needed in order to improve its robustness and convergence speed. The performance of different genetic operators could be compared, for instance, or even the use of an optimization method other than the genetic algorithm could be considered in case it is better suited to solve such combinatorial problems. As a result of this study, the scheduling algorithm may be able to achieve greater generations' makespan reductions at a lower cost.

The static and dynamic versions of the task management algorithm in which the aforementioned scheduler is contained have also some improvement possibilities and are outlined in the next three paragraphs.

The first aspect to be improved is the method used for calculating the time limitation for the task scheduling algorithm. According to the author's mind, a good approach could be the simultaneous use of 2 criteria, being the scheduling time limited in each case by the most restrictive criterion: i) a limitation depending on the previous generation's makespan (as it was implemented for this Doctoral Thesis), and ii) a limitation depending on the size of the combinatorial problem to be solved, as it is thought that having the combinatorial optimization problems a finite solutions space, the time required by a certain search method to find a good solution could be standardized for different problem sizes. Such a combined criterion may also be useful for balancing hybrid optimizations, in which the size of the batches of individuals created by the global and the local methods may differ significantly. Moreover, it could also be used when the scheduler is run several times per generation by the dynamic task manager, as the number of individuals to be scheduled is progressively decreased.

A second aspect to be improved is the CPU usage during the scheduling process. In the proposed algorithms the scheduler is always run in a single processor while other processors remain idle waiting for the scheduler to finish. Some alternatives to avoid wasting computational time in this manner could be to either parallelize the execution of the scheduler or to run several schedulers simultaneously and use the best among the obtained task distributions. In case a "dedicated master" processor partitioning model was used, the scheduler could be uninterruptedly run in the master processor with the most updated information and provide the best available schedule when required by

any other processor.

A third aspect would consist on finding an automated method able to set the optimal configuration of the task management algorithm regarding the number of schedulers to be run per generation and the percentage defining the maximal time limitation for each scheduler run based on the previous generation's makespan.

Finally, note that hardware homogeneity was assumed in every case studied in the scope of this Doctoral Thesis. Nevertheless, hardware heterogeneity cannot be avoided in massively parallel applications and it involves variable inter-processor communication delays and different data processing speeds. These factors need to be taken into consideration by the task management algorithm to optimally map tasks to processors, being the complexity of the optimization problem to be solved substantially increased.