Chapter 1
Parallel Evolutionary Computation Framework for Single- and Multiobjective Optimization

Bogdan Filipič and Matjaž Depolli

Abstract

Evolutionary computation is an area of computer science utilizing the mechanisms of biological evolution in computer problem solving. It is concerned with theoretical studies, design and application of stochastic optimization procedures, known as Evolutionary Algorithms (EAs). EAs have proven effective and robust in solving demanding optimization problems that are often difficult if not intractable to traditional numerical methods. They are nowadays widely applied in science, engineering, management and other domains. However, a drawback of EAs is their computational complexity which originates from iterative population-based search of the solution space. On the other hand, processing a population of candidate solutions makes EAs amenable to parallel implementation that may result in significant calculation speedup.

This chapter presents a parallel evolutionary computation framework developed for solving numerical optimization problems with one ore more objectives, and evaluates its performance on a high-dimensional optimization task from industrial practice. The chapter starts with an introduction to optimization problems. It distinguishes between single- and multiobjective optimization and reviews the concepts needed to deal with multiobjective optimization problems, such as the dominance relation and Pareto optimality. Next, EAs as a general-purpose optimization method are described, with a focus on Differential Evolution (DE) which is a particular kind of EA used in our framework. Then, parallelization of EAs is discussed in view of known parallelization types and speedup calculation. The chapter continues with an introduction to the optimization problem in industrial continuous casting, used as a

Bogdan Filipič
Department of Intelligent Systems, Jožef Stefan Institute, Jamova cesta 39, 1000 Ljubljana, Slovenia, e-mail: bogdan.filipic@ijs.si

Matjaž Depolli
Department of Communication Systems, Jožef Stefan Institute, Jamova cesta 39, 1000 Ljubljana, Slovenia, e-mail: matjaz.depolli@ijs.si

test problem in this work. Afterwards, the proposed parallel evolutionary computation framework is presented. The framework is based on DE and implemented on a cluster of personal computers. It is evaluated on single- and multiobjective variants of the casting optimization problem and the results analyzed from the perspective of the problem domain and, in particular, the achieved speedup.

1.1 Introduction

In the last decades, a number of computational techniques have been proposed that take inspiration from natural phenomena. Among them is evolutionary computation [1, 2] with the underlying idea of employing the mechanisms of biological evolution in computer problem solving. Search and optimization algorithms designed according to these principles, known as Evolutionary Algorithms (EAs), simulate the evolution of candidate solutions to a given problem, usually starting from a randomly created initial set, and iteratively improving its members until their convergence. Despite its simplicity, this approach has proved efficient and widely applicable. EAs can nowadays be found in a variety of application domains, ranging from science [3] to engineering [4] to management [5].

EAs are in many respects superior to traditional algorithms. Candidate solutions in an EA can be represented and varied in a number of ways which makes these algorithms suitable for solving radically different types of optimization problems. Their operation relies on the quality of solutions being processed and requires no additional information about the search space. As a result, noncontinuous, multimodal and time-dependent problems, hard to solve with traditional algorithms, can be successfully approached with EAs. On the other hand, the population-based search performed with EAs, as opposed to the single-point search in most other algorithms, has both advantages and disadvantages. On the positive side, it results in more than one solution produced in a single algorithm run, which provides a user with alternatives that are sometimes highly desirable. As a disadvantage comes the computational burden of processing a population of candidate solutions. What helps here is the inherent parallelism of EAs: the solutions can be evaluated independently and thus run in parallel for the entire population. This property makes EAs amenable to parallel implementation that may significant speedup the calculation. This is particularly useful when solution evaluation is computationally expensive, which is often the case with real-world problems.

This chapter describes a parallel evolutionary computation framework developed for solving numerical optimization problems. It starts with a formal introduction to optimization problems and distinguishes between single- and multiobjective optimization. It presents the basic concepts needed to deal with multiobjective optimization problems, such as the dominance relation and Pareto optimality. It continues with a presentation of EAs in general and then focuses on Differential Evolution (DE), an EA specialized in numerical optimization. Both the original single-objective DE and its multiobjective extension are outlined. Next, paralleliza-

tion of EAs is discussed regarding the types of parallelization and the calculation of speedups. The chapter then introduces the task of process parameter tuning in industrial continuous casting of steel where the goal is to satisfy the empirical metallurgical criteria formulated to increase the quality of cast steel [6]. This problem will later be used to evaluate the proposed evolutionary computation framework. The framework itself is explained in detail. It makes use of any number of processors available and increases the performance of the optimization procedure by distributing the evaluation of candidate solutions among the processors. Installed on a cluster [7] of Opteron computers running under Linux, it is empirically evaluated on the casting optimization problem. Both single- and multiobjective variants of the problem are exercised and the results analyzed in view of the problem domain and, in greater detail, the achieved calculation speedup. The optimization results are comparable to the results obtained previously on the same problem instances, while, in accordance with predictions, high speedups are achieved. These findings also suggest further work to enhance the performance of the parallel framework on hardware architectures different from the one used in this work.

1.2 Optimization Problems

Numerous tasks in science, engineering and business require finding the best solution from a set of candidate solutions that can be evaluated according to a quality measure and have to satisfy various constraints. These tasks are called optimization problems, and the procedure of solving an optimization problem is optimization.

We focus on numerical optimization problems where candidate solutions are vectors of real decision variables (sometimes called problem parameters)

$$\mathbf{x} = [x_1, x_2, ..., x_n]^{\mathrm{T}},$$

and the quality measure is a real function $f(\mathbf{x})$ defined over $\mathbb{R}^n$. Formally, a numerical optimization problem is to find a vector

$$\mathbf{x}^* = [x_1^*, x_2^*, ..., x_n^*]^{\mathrm{T}}$$

that fulfills boundary constraints

$$x_i^{\mathrm{low}} \leq x_i \leq x_i^{\mathrm{up}}, \;\; i = 1, 2, ..., n,$$

inequality constraints

$$g_j(\mathbf{x}) \geq 0, \;\; j = 1, 2, ..., J,$$

and equality constraints

$$h_k(\mathbf{x}) = 0, \;\; k = 1, 2, ..., K,$$

and optimizes $f(\mathbf{x})$.

The boundary constraints restrict each decision variable x_i to take values within its lower bound x_i^{low} and upper bound x_i^{up}, and determine a *decision variable space* (or decision space, for short) of a numerical optimization problem. Solutions satisfying all boundary constraints, inequality constraints, and equality constraints are called *feasible solutions*. On the other hand, solutions not satisfying all the constraints are *infeasible*. Furthermore, $f(\mathbf{x})$ is known as the *objective function* or cost function. Optimizing $f(\mathbf{x})$ means either minimizing or maximizing it.

Note that the objective function is not always given explicitly. Particularly in practical optimization problems it may be very demanding, if not impossible, to formulate it. Alternatively, candidate solutions can be evaluated empirically through experiments, measurements, computer simulation, etc.

The traditional definition of a numerical optimization problem given above assumes there is only one objective, and solving such a problem is therefore referred to as single-objective optimization. However, most real-world optimization problems involve multiple objectives, and these are often in conflict with each other in the sense that improvement of a solution with respect to a selected objective deteriorates it with respect to other objectives. In such cases we deal with multiobjective optimization problems. These can be formally stated analogously to the single-objective ones with the exception that the task is now to optimize a vector function

$$\mathbf{f}(\mathbf{x}) = [f_1(\mathbf{x}), f_2(\mathbf{x}), ..., f_M(\mathbf{x})]^{\mathrm{T}}.$$

As a result, there are two spaces associated with a multiobjective optimization problem: in addition to an N-dimensional decision variable space, there is an M-dimensional *objective space* where the objective vectors can be partially ordered using the *dominance relation*. Objective vector $\mathbf{x}$ is said to dominate objective vector $\mathbf{y}$, formally $\mathbf{x} \prec \mathbf{y}$, iff $\mathbf{x}$ is not worse than $\mathbf{y}$ in all objectives and is better than $\mathbf{y}$ in at least one objective.

Let us illustrate the dominance relation with an example. Consider a multiobjective optimization problem with two objectives, f_1 and f_2, that both need to be minimized. Fig. 1.1 shows five solutions to this problem in the objective space. Comparing solution **a** with other solutions, we can observe that **a** dominates **b** since it is better than **b** in both objectives, i.e., $f_1(\mathbf{a}) < f_1(\mathbf{b})$ and $f_2(\mathbf{a}) < f_2(\mathbf{b})$. It also dominates **c** as it is better than **c** in objective f_2 and not worse in objective f_1. On the other hand, **d** outperforms **a** in both objectives, therefore **d** dominates **a** or, in other words, **a** is dominated by **d**. However, regarding **a** and **e**, no such conclusion can be made because $f_1(\mathbf{a}) < f_1(\mathbf{e})$ and $f_2(\mathbf{a}) > f_2(\mathbf{e})$. We say that **a** and **e** are incomparable.

In general, in a set of solutions to a multiobjective optimization problem, there is a subset of solutions that are not dominated by any other solution (**d** and **e** in the example from Fig. 1.1). Referring to the decision variable space, we call this subset a *nondominated set of solutions*, and in the objective space the corresponding vectors are called a *nondominated front of solutions*. The concept is illustrated in Fig. 1.2 where both objectives need to be minimized again. The nondominated set of the entire feasible search space is known as the *Pareto optimal set*, and the non-

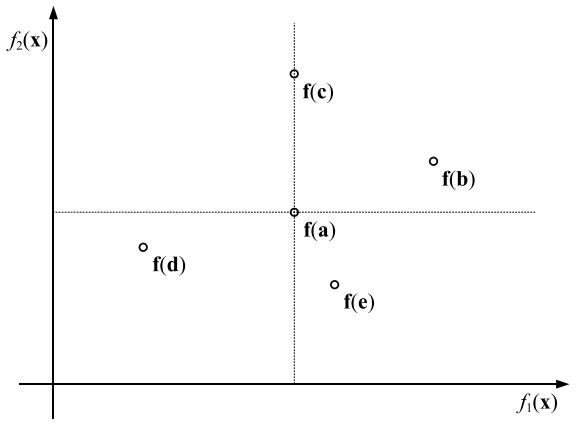

Fig. 1.1 Comparison of solutions to a multiobjective optimization problem in the objective space.

dominated front of the entire feasible search space the *Pareto optimal front* (named after Vilfredo Pareto (1848–1923), an Italian economist, sociologist, and a pioneer in the field of multiobjective optimization).

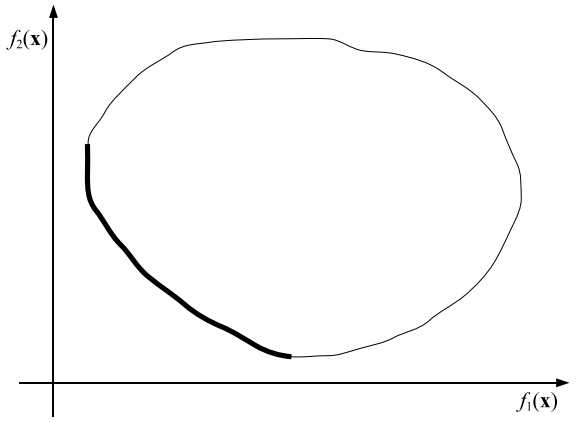

Fig. 1.2 Nondominated front of solutions in the objective space (both objectives need to be minimized).

Objective vectors from the Pareto optimal front represent different trade-offs between the objectives, and without additional information no vector can be preferred to another. With a multiobjective optimizer we search for an *approximation set* that approximates the Pareto optimal front as closely as possible. In practical multiobjective optimization it is often important to provide a diverse choice of trade-offs. Therefore, besides including vectors close to the Pareto optimal front, the approximation set should also contain near-optimal vectors that are as diverse as possible.

1.3 Evolutionary Algorithms

Evolutionary Algorithms (EAs) is a common name for a family of search and optimization procedures created and studied in the field of evolutionary computation [1, 2]. The underlying idea is to solve a given problem through computer simulated evolution of candidate solutions. The set of candidate solutions processed by an EA is called a population, and the population members are referred to as individuals. They are represented in the form suitable for solving a particular problem. Often used representations include bit strings, real-valued vectors, permutations, tree structures and even more complex data structures. In addition, a fitness function needs to be defined that assigns a numerical measure of quality to the individuals; it roughly corresponds to the cost function in optimization problems.

An EA, shown in pseudocode as Algorithm 1, starts with a population of randomly created population members, and iteratively improves them by employing evolutionary mechanisms, such as survival of the fittest individuals and exchange of genetic information between the individuals. The iterative steps are called generations, and in each generation the population members undergo selection and variation.

Algorithm 1 Evolutionary Algorithm (EA)

```
create the initial population ℙ of random solutions;
evaluate the solutions in ℙ;
while stopping criterion not met do
    create an empty population ℙ_new;
    repeat
        select two parents from ℙ;
        create two offspring by crossing the parents;
        mutate the offspring;
        evaluate the offspring;
        add the offspring into ℙ_new;
    until ℙ_new is full;
    copy ℙ_new into ℙ;
end while
```

The selection phase of the algorithm is an artificial realization of the Darwinian principle of survival of the fittest among individuals. The higher the fitness of an individual (i.e., the quality of a solution), the higher the probability of participating in the next generation. In the variation phase, the individuals are modified in order to generate new candidate solutions to the considered problem. For this purpose, the EA applies operators, such as crossover and mutation, to the individuals. The crossover operator exchanges randomly selected components between pairs of individuals (parents), while mutation alters values at randomly selected positions in the individuals.

The algorithm runs until a stopping criterion is fulfilled. The stopping criterion can be defined in terms of the number of generations, required solution quality or as

a combination of both. The best solution found during the algorithm run is returned as a result.

EAs exhibit a number of advantages over traditional specialized methods and other stochastic algorithms. Besides the evaluation of candidate solutions, they require no additional information about the search space properties. They are a widely applicable optimization method, straightforward for implementation and suitable for hybridization with other search algorithms. Moreover, it is not difficult to incorporate problem-specific knowledge into an EA in the form of specialized operators when such knowledge is available. Finally, by processing populations of candidate solutions, they are capable of providing alternative solutions to a problem in a single algorithm run. This is extremely valuable when solving multimodal, time-dependent and multiobjective optimization problems.

A somewhat more specialized EA is Differential Evolution (DE) [8, 9]. It was designed for solving numerical optimization and has proved very efficient in this problem domain. In DE, candidate solutions are encoded as n-dimensional real-valued vectors. As outlined in Algorithm 2, new candidates are constructed through operations such as vector addition and scalar multiplication (in line 8, F denotes a predefined scalar value). After creation, each candidate is evaluated and compared with its parent and the best of them is added to the new population.

Algorithm 2 Differential Evolution (DE)

```
 1: create the initial population ℙ of random solutions;
 2: evaluate the solutions in ℙ;
 3: while stopping criterion not met do
 4:     create an empty population ℙ_new;
 5:     repeat
 6:         for each solution P_i, i = 1..pop_size from ℙ do
 7:             randomly select three different solutions I_1, I_2, I_3 from ℙ;
 8:             create a candidate solution C := I_1 + F · (I_2 − I_3);
 9:             alter C by crossover with P_i;
10:             evaluate C;
11:             if C is better than P_i then
12:                 add C into ℙ_new
13:             else
14:                 add P_i into ℙ_new;
15:             end if
16:         end for
17:     until ℙ_new is full;
18:     copy ℙ_new into ℙ;
19: end while
```

1.3.1 Multiobjective Evolutionary Algorithms

In multiobjective optimization, finding an approximation of the Pareto optimal front in a single run requires a population-based method. Therefore, EAs are a reasonable choice for this task. However, since the objective space in multiobjective optimization problems is multidimensional, any EA originally designed for single-objective optimization needs to be extended to deal with multiple objectives. This has been done with several EAs that are now used as multiobjective optimizers and referred to as Multiobjective Evolutionary Algorithm (MOEAs) [10, 11, 12].

Based on the single-objective DE is Differential Evolution for Multiobjective Optimization (DEMO) [13]. It extends DE with a particular mechanism for deciding which solutions to keep in the population (see Algorithm 3). For each parent in the population, DEMO constructs a candidate solution in the same way as DE. If the candidate dominates the parent, the candidate is added to the new population. If the parent dominates the candidate, the parent is added to the new population. Otherwise, if the candidate and its parent are incomparable, they are both added to the new population. During the construction of candidates for all parents in the population, the new population possibly increases. In this case, it is truncated to the original population size using nondominated sorting and the crowding distance metric in the same manner as in the NSGA-II multiobjective algorithm [14]. These steps are repeated until a stopping criterion is met.

The serial versions of DE and DEMO described here will be used as a foundation for our parallel evolutionary computation framework to efficiently deal with single- and multiobjective optimization problems, respectively.

1.4 Parallel Single- and Multiobjective Evolutionary Algorithms

EAs are an example of inherently parallel algorithms. Fitness evaluation can be independently calculated for each individual and therefore run in parallel for the entire population at a time. This mainly results in a faster algorithm execution, i.e., speedup [15], although it could in some cases also loosen hardware bottlenecks, such as memory shortage. This chapter focuses on the speedup, but also provides notes on efficiency (speedup normalized with the number of processors) and hardware bottlenecks where applicable.

1.4.1 Parallelization Types

There are four types of parallel EAs [16, 17], three basic: *master-server* (also called *global parallelization*), *island*, *diffusion* (also known as *cellular*), and *hybrid* that encompasses combinations of the basic types.

Algorithm 3 Differential Evolution for Multiobjective Optimization (DEMO)

```
 1: create the initial population ℙ of random solutions;
 2: evaluate the solutions in ℙ;
 3: while stopping criterion not met do
 4:     create an empty population ℙ_new;
 5:     repeat
 6:         for each solution P_i, i = 1..pop_size from ℙ do
 7:             randomly select three different solutions I_1, I_2, I_3 from ℙ;
 8:             create a candidate solution C := I_1 + F·(I_2 − I_3);
 9:             alter C by crossover with P_i;
10:             evaluate C;
11:             if C dominates P_i then
12:                 add C into ℙ_new
13:             else
14:                 if P_i dominates C then
15:                     add P_i into ℙ_new;
16:                 else
17:                     add both P_i and C into ℙ_new;
18:                 end if
19:             end if
20:         end for
21:         if ℙ_new contains more than pop_size solutions then
22:             truncate ℙ_new;
23:         end if
24:     until ℙ_new is full;
25:     copy ℙ_new into ℙ;
26: end while
```

Master-slave EAs are the most straightforward type of parallel EAs and the only one that makes use of the EAs inherent parallelism. As a consequence, they traverse the search space identically to their serial counterparts. A master-slave EA can be visualized as a master node running a serial EA with a modification in fitness evaluation. Instead of evaluating fitness serially, one individual at a time, until the entire population is evaluated, individuals are evaluated on the master and slave nodes in parallel. The highest efficiency of this parallelization type can be achieved on computers with homogeneous processors and in problem domains where the fitness evaluation time is constant and independent of the individual. When these criteria are fulfilled and the fitness evaluation time is long compared to the time required for other parts of the algorithm, near-linear speedup is possible.

Island EAs, in contrast, are multiple-population algorithms, consisting of several largely independent subpopulations that occasionally exchange a few individuals. In an island EAs, each processing node represents an island, running a serial EA on a subpopulation. A new operator is introduced – migration, that handles the exchange of individuals between the islands. Migration occurs either in a predefined intervals, e.g., every several generations, or after special events, e.g., when subpopulations start to converge. Communication overhead is therefore smaller compared to the master-slave parallelization type. In general, speedup increases with the number of

islands, but the overall efficiency depends on how well the problem is suited for solving with multiple-population EAs compared to single-population EAs.

Diffusion EAs split population into multiple small subpopulations and divide them among the processing nodes. Every subpopulation is allowed to communicate (individuals may interact) with a predefined neighborhood of other subpopulations. These algorithms can also be considered single-population with structurally constrained interactions between individuals. Parallelization of this type has large communication overhead and may be worth considering only on large computer clusters with dedicated interconnections between the neighboring processing nodes. Speedup and efficiency depend greatly on the properties of interconnections and the suitability of the problem to the structural constraints imposed by the algorithm.

Hybrid parallel EAs are an attempt to minimize the weaknesses of the basic type algorithms through their hierarchic composition. For example, the island type may be implemented on top of the master-server type, providing possibility to use all available processing nodes, while keeping the number of islands variable. Hybrid EAs are very adaptable to the underlying hardware architecture, but their design and implementation are more complex.

1.4.2 Calculation of Speedups

Traditionally, speedup is defined as the ratio between the execution times of the best serial algorithm and the best parallel algorithm:

$$S = \frac{T_s}{T_p} \,. \tag{1.1}$$

As this definition depends on the execution times, we call it the measured speedup, to contrast it with the estimated speedup. In case of the master-slave EAs, selection of the best algorithms is trivial, since the parallel algorithm traverses the search space identically to its serial counterpart. Therefore, for a valid speedup measurement, both algorithms should be run with the same algorithm parameter setting, for the same number of generations.

More care should be taken when dealing with other types of parallel EAs. Modifications needed for the island and diffusion EAs may have a positive influence on some EAs and in some problem domains. These modifications can always be translated back into a serial algorithm, since every parallel algorithm can be trivially serialized. This way, a new best known serial algorithm for calculation of speedup can be obtained. Therefore, the best serial counterpart to a particular multipopulation parallel EA may either be its serial implementation, or the original, single-population EA.

The only limiting factor for serialization could be hardware (for example, multiple-population EAs require more memory than single-population EAs). In such a case, parallelization serves as a means of alleviating hardware constraints as well. The

obtained speedup in such cases would be due to parallel execution and due to algorithm improvements, with either contribution unobtainable from the measurements alone.

Additionally, the island and diffusion EAs make use of additional parameters – the number of subpopulations, and the size and shape of the neighborhood. These parameters are in parallel implementations to a large extent fixed to the number of processors and the computer architecture, but are free in serial implementations. Therefore, the best algorithm parameter setting may differ between serial and parallel implementations.

While measuring the parallelization speedup of the master-slave EAs is straightforward, it requires a lot of additional work for the multipopulation parallel EAs. Since the knowledge of speedup is usually not a priority to the algorithm developers, the parallel multipopulation EAs are often compared only to the original serial EAs. This technique frequently yields super-linear speedups, which are a good indication of the use of suboptimal serial algorithms.

We explore the master-slave EAs in more detail, to estimate their limitations in speedup. We start with the theoretical limit on speedup according to the Amdahl's law:

$$S_{max} = \frac{1}{(1-P) + \frac{P}{N}} , \tag{1.2}$$

where P is the parallel portion of the algorithm and N is the number of processors. The actual speedup of an algorithm will depend on how well the parallel portion can be spread up among N processors. Considering the simplest master-slave parallelization type, where only fitness evaluations are parallelized, P is the portion of the serial algorithm execution time spent on fitness evaluation. It should be noted that through the process of parallelization, the interprocessor communication is added to the algorithm, which effectively decreases its parallel portion. As demonstrated later on, when the interprocessor communication is taken into consideration, P can still reach very high values if fitness evaluation is complex and time consuming. On the other hand, N is limited by the population size N_p. Only the population of a single generation can be evaluated at a time, even when more processors are available. Speedup upper bound therefore equals the population size:

$$\lim_{P \to 1} S_{max} = \lim_{P \to 1} \frac{1}{(1-P) + \frac{P}{N_p}} \; = \; N_p \; . \tag{1.3}$$

Another important observation is that not only should $N_p \leq N$, but also $N_p \mid N$ (N_p divides N), for the algorithm to fully utilize all processors. The algorithm needs $\lceil \frac{N_p}{N} \rceil$ iterations to fully evaluate the population and therefore has $\lceil \frac{N_p}{N} \rceil \times N$ processor time slots to fill with N_p tasks (fitness evaluations). It is free to choose the best way to allocate the tasks to processor time slots over the iterations but there will always remain $N_p \bmod N$ unallocated slots per generation, for which the processors will be left idle. From this we can derive the effective number of processors used by the algorithm $N_{\text{eff}} = N_p / \lceil \frac{Np}{N} \rceil$. Finally, substituting N with N_{eff} in Eq. (1.2) we can

rewrite the speedup equation as

$$S_{\max} = \frac{1}{(1-P) + \frac{P \times \lceil \frac{N_p}{N} \rceil}{N_p}} \,. \tag{1.4}$$

An example of $S_{\max}(N)$ for population size $N_p = 32$ and parallel fraction $P = 1$ is shown in Fig. 1.3.

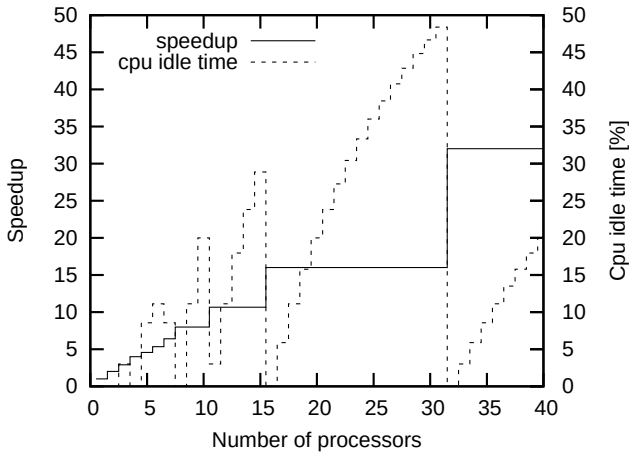

Fig. 1.3 Maximum speedup and processor idle time vs. the number of available processors for a master-slave parallel EA with $P \to 1$ and $N = 32$.

The dependence of speedup on the number of processors is alleviated by the insensitivity of EAs to the population size. Because of the stochastic nature of EAs, an approximate interval can be determined rather than an exact number for the best population size on a given problem. If the interval is larger than the number of processors, then fixing the population size to a multiple of the number of processors while keeping it inside the interval is possible. In cases when optimal selection of the population size within the interval is not possible, speedup calculation should be amended. Suppose an EA with the optimal population size in comparison to an EA with the selected population size has speedup S_{opt}. The actual maximum speedup of a parallel master-server EA will then be

$$S^*_{\max} = \frac{S_{\max}}{S_{\text{opt}}} \,. \tag{1.5}$$

1.5 Casting Process Optimization Task

Continuous casting of steel is widely used at modern steel plants to produce various steel semi-manufactures. The proces is schematically shown in Fig. 1.4. In this process, liquid steel is poured into a bottomless mold which is cooled with internal water flow. The cooling in the mold extracts heat from the molten steel and initiates the formation of a solid shell. The shell formation is crucial for the support of the slab behind the mold exit. The slab then enters the secondary cooling area where additional cooling is performed by water sprays. Led by the support rolls, the slab gradually solidifies and finally exits the casting device. At this stage it is cut into pieces of predefined length.

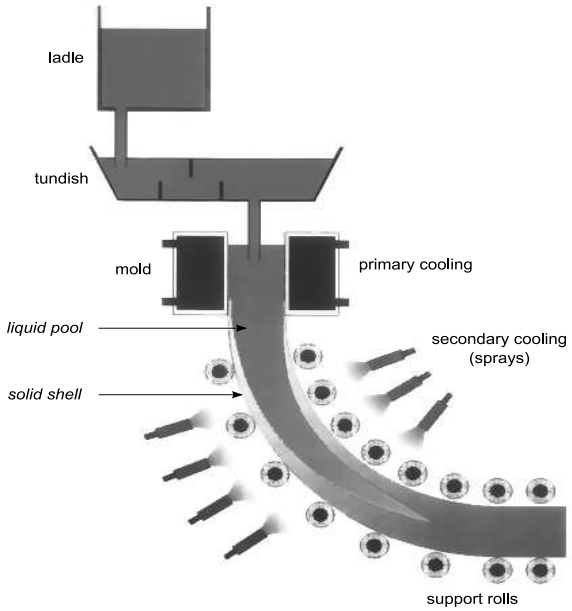

Fig. 1.4 A schematic view of continuos casting of steel.

The secondary cooling area of the casting device is divided into cooling zones and the cooling water flows in the zones can be set individually. In each zone, cooling water is dispersed to the slab at the center and corner positions. Target temperatures are specified for the slab center and corner in every zone and the optimization task is to tune the cooling water flows in such a way that the resulting slab surface temperatures match the target temperatures as closely as possible. From metallurgical practice this is known to reduce cracks and inhomogeneities in the structure of the cast steel. Formally, an objective f_1 is introduced to measure deviations of actual temperatures from the target ones:

$$f_1 = \sum_{i=1}^{N_Z} |T_i^{\text{center}} - T_i^{\text{center}*}| + \sum_{i=1}^{N_Z} |T_i^{\text{corner}} - T_i^{\text{corner}*}|, \tag{1.6}$$

where N_Z denotes the number of zones, T_i^{center} and T_i^{corner} the slab center and corner temperatures in zone i, and $T_i^{\text{center}*}$ and $T_i^{\text{corner}*}$ the respective target temperatures in zone i. This objective encompasses the key requirement for the process to results in high-quality cast steel. Technically, this is a single-objective version of the casting optimization task.

In addition, there is a requirement for core length, l^{core}, which is the distance between the mold exit and the point of complete solidification of the slab. The target value for the core length, $l^{\text{core}*}$, is prespecified, and the actual core length should be as close to it as possible. Shorter core length may result in unwanted deformations of the slab as it solidifies too early, while longer core length may threaten the process safety. This requirement can be treated as an additional objective, f_2:

$$f_2 = |l^{\text{core}} - l^{\text{core}*}|, \tag{1.7}$$

and the more demanding version of the optimization task is then to minimize both f_1 and f_2 over possible cooling patterns (water flow settings). The two objectives are conflicting, hence it is reasonable to handle this optimization problem in the multiobjective manner.

In the optimization procedure, water flows cannot be set arbitrarily, but according to the technological constraints. For each zone, lower and upper bounds are prescribed for the center and corner water flows. Moreover, to avoid unacceptable deviations of the core length from the target value, a hard constraint is imposed: $f_2 \leq \Delta l_{\max}^{\text{core}}$. Solutions violating the water flow constraints or the core length constraint are considered infeasible.

A prerequisite for optimization of this process is an accurate mathematical model of the casting process, capable of calculating the temperature field in the slab as a function of coolant flows and evaluating it with respect to the objectives given by Eqs. (1.6) and (1.7). For this purpose we use a numerical simulator of the process with the Finite Element Method (FEM) discretization of the temperature field and the related nonlinear heat transfer equations solved with relaxation iterative methods [6].

1.6 Parallel Evolutionary Computation Framework

We present a parallel framework for numerical single- and multiobjective optimization on homogeneous parallel computer architectures. It is based on single-objective Differential Evolution (DE) and is extended to Differential Evolution for Multiobjective Optimization (DEMO) when multiobjective optimization is required.

The framework is able to utilize any number of processors by implementing the master-slave parallelization scheme for both optimization algorithms. Although de-

signed for use on homogeneous parallel computer architectures, it can use heterogeneous architectures as well, but with lower utilization of faster processors. When a single processor is used, master-slave algorithms degenerate into their non-parallel versions, thus avoiding potential overhead of the parallelization scheme.

In the framework, the optimization procedure is performed in three stages: initialization, generational computation, and finalization. The initialization consists of reading the input files and settings, and the setup of initial population. Generational computation iterates over generations, where in each iteration fitness values are calculated for individuals of the current population and the evolutionary algorithm operators are applied to them, spawning the next generation. In finalization, the results are formatted and returned to the user.

While the initialization and finalization are run by the master process, the generational computation can be run in parallel by all processes. Each iteration starts with the master process holding a vector of individuals of unknown fitness. These are then evaluated by the master and slave processes in parallel, which requires interprocess communication. For this purpose, the Message Passing Interface (MPI) [18] is used. It implements the interprocess communication in a two-part, coupled fashion. The first part distributes the data on the individuals among the slave processes, and the second part returns the fitness values to the master process. For the sake of simplicity, only the data on one individual is transferred to each slave process per communication couple. This forces the communication couple to happen more than once per generation if the population size is larger than the number of processors. The part in which the master process receives the results from the server processes is also blocking, i. e., it waits for all the results before it continues execution, effectively synchronizing the processors. This, in combination with multiple communication couples per generation, causes some unnecessary synchronizations. After the fitness values for all individuals are known, the master process applies the evolutionary algorithm operators and spawns the next generation. Slave processes are idle at this time, waiting to receive the data on individuals of the next generation.

The parallelization approach employed by the proposed framework is, in the context of multiobjective optimization, known as the Parallel Function Evaluation (PFE) variant of the single-walk parallelization [19]. It is aimed at speeding up the computations, while the basic behavior of the underlying algorithms remains unchanged.

1.6.1 Speedup Estimation

What is the expected speedup of the framework running on several processors in comparison to the framework running on a single processor, solving an optimization problem? One should be able to answer this question before starting the optimization, to use the most appropriate number of processors. To answer this question, we start with the speedup as defined in Eq. (1.1). We simplify it by only using the time for generational computation instead of the total execution time for both,

the serial and parallel implementations. This is reasonable because the initialization and finalization are faster than even a single application of the evolutionary algorithm operators, and are negligible in cases when parallelization is considered, that is, when the total execution time is expected to be long. Furthermore, because the generational computation is a series of identical single generation computations, we simplify the definition of speedup to only consider a single generation. Thus we get the initial form of the speedup equation:

$$S = \frac{T_s + T_e * N_p}{T_s + T_e \times \lceil \frac{N_p}{N} \rceil} , \tag{1.8}$$

where T_e is the time required for a single fitness evaluation, T_s is the time required for the execution of a single generation, excluding the time required for fitness evaluations, N_p is the population size, and N is the number of processors. This is a good estimation if two criteria are met. The first criterion is constant time of fitness evaluation. This means that all fitness evaluations take exactly the same amount of time to complete, not depending on the input, the processor, nor any random factor. The second criterion is that parallelization produces negligible calculation overhead. In the master-slave parallelization scheme, the overhead consists of the time required for interprocess communication, including the time the master process is waiting for the results from the slave processes.

The time required for communication, T_c, can be simply added to the denominator in Eq. (1.8). It is irrelevant when it is orders of magnitude shorter than the fitness evaluation time, but when it is not, it has to be estimated, because it depends on the problem domain as well as the communication protocols and hardware. For instance, first the number of bytes used to represent the fitness function input parameters sets the base size of messages sent from the master to the slaves, and the number of bytes used to represent the evaluation results sets the base size of messages sent from the slaves to the master. Then the protocols over which the messages are sent, e.g., TCP/IP, and the library which implements message passing, e.g., MPI, increase the message sizes with their overhead. Last, the hardware determines how fast the messages of certain sizes can be sent between the processors. The speedup then equals to:

$$S = \frac{T_s + T_e * N_p}{T_c + T_s + T_e \times \lceil \frac{N_p}{N} \rceil} \tag{1.9}$$

Eliminating the constant fitness evaluation time criterion from the equation is more complex. The master process cannot apply the evolutionary algorithm operators, until all the individuals of the population have their fitness values evaluated. The process executing the longest fitness evaluation thus forces all other processes to wait until it finishes. We define the time required for execution of n fitness evaluations in parallel ($T_{\mathrm{ep}}(n)$) in Eq. (1.10) as the expected value of maximum of n independent fitness evaluation times. One way of calculating the expected value is numerically, from the cumulative distribution function (CDF) of maximum time of

n fitness evaluations, which equals the CDF of fitness evaluation time, raised to the power of n.

$$T_{\text{ep}}(n) = E(\max_{i=1}^{n}\{t_{e,i}\}) \tag{1.10}$$

The framework executes a series of parallel evaluations during a single generation if the population size is larger than the number of processors. Individuals are split into $\lceil \frac{N_p}{N} \rceil$ groups, with first $\lfloor \frac{N_p}{N} \rfloor$ groups of the size equal to the number of processors, and the last group (if $\lceil \frac{N_p}{N} \rceil \neq \lfloor \frac{N_p}{N} \rfloor$) of size $N_p \bmod N$. Each group is separately evaluated in parallel, adding to the total evaluation time of a population, which can now be calculated as $T_{\text{ep}}(N) \times \lfloor \frac{N_p}{N} \rfloor + T_{\text{ep}}(N_p \bmod N)$. The final form of the estimated speedup equation can now be written as:

$$S = \frac{T_s + T_e * N_p}{T_c + T_s + T_{\text{ep}}(N) \times \lfloor \frac{N_p}{N} \rfloor + T_{\text{ep}}(N_p \bmod N)} \tag{1.11}$$

1.7 Empirical Evaluation

An empirical evaluation of the proposed framework was performed on the computer cluster comprised of 17 dual processor computers. Optimization of continuous casting served as a test domain for both the single- and multiobjective optimization.

1.7.1 Experimental Setup

For the evaluation of the framework, a cluster of 17 dual-processor nodes (each node being a personal computer) was used. The nodes are all interconnected through an Ethernet switch, and, in addition, there are several direct interconnections between the nodes (see Fig. 1.5). Nodes 1 through 16 are connected by a toroidal 4-mesh, and nodes 1 through 4 are directly connected to the additional node. This node serves as a host node, through which users access the cluster. Static routing is used to direct the communication between the pairs of nodes, which are not physically interconnected, through the switch. This makes the use of any desired topology possible. In our tests, star topologies of various sizes were used.

The cluster is composed of identical personal computers, each containing two AMD Opteron 244 processors, 1024 MB of RAM, a hard disk drive, and six 1000 MB/s Full Duplex Eethernet ports. On each computer, there is as an independent installation of the Fedora Core 2 operating system and the MPICH v1.2.6 library that supports communication between the computers and is an implementation of the Message Passing Interface (MPI). During the experiments, all nodes are required to

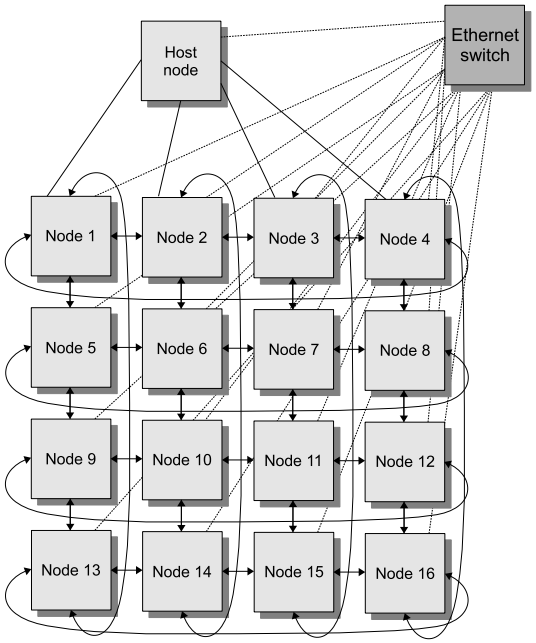

Fig. 1.5 Architecture of the cluster used in tests.

be running only the background system processes which leaves nearly all capabilities to be used by the framework.

The parallel optimization algorithm was written in C++ and compiled with *gcc v3.3.3* for target 64-bit *Linux*, while the continuous casting simulator was compiled for 32-bit *Microsoft Windows* and was executed through an early version of *Wine* (an application providing the compatibility layer for the *Microsoft Windows* programs). There was also a layer of scripts, translating the communication between the optimization algorithm and the simulator, i.e., filtering and converting input/output files of the simulator.

1.7.2 Experiments and Results

Numerical experiments in optimizing the continuous casting process were performed to analyze both the effectiveness and efficiency of the developed parallel framework. The former relates to the quality of results, while the latter refers to the speedup achieved with the parallel optimization approach.

Optimization calculations were performed for a selected steel grade and slab cross-section of 1.70 m $\times$ 0.21 m and for various casting speeds: the usually practised speed of 1.8 m/min, and two lower speeds of 1.6 m/min and 1.4 m/min that are exercised when the process needs to be slowed down to ensure the continuity of casting, for example, when a new batch of molten steel is delayed. Candidate

solutions in parallel DE and DEMO were encoded as 18-dimensional real-valued vectors, representing coolant flow values at the center and the corner positions in the nine zones of the secondary cooling area. Search intervals for coolant flows at the center and the corner positions in zones 1–3 were between 0 and 50 m^3/h, and in the zones 4–9 between 0 and 10 m^3/h. The target core length, $l^{\text{core}*}$, was 27 m and the maximum allowed deviation from the target, $\Delta l^{\text{core}}_{\text{max}}$, was 7 m. Reasonable population size found in initial experiments was 30.

It turned out that for the single-objective and the two-objective versions of the task the parallel optimization procedure was able to discover the solutions known from previous applications of serial optimization algorithms [6, 20]. To illustrate the results for the more challenging two-objective version, Fig. 1.6 shows the resulting nondominated fronts of solutions (approximating Pareto optimal fronts) found by the parallel DEMO algorithm for various casting speeds. It can be seen that the two objectives can simultaneously be fulfilled to the highest degree at the regular casting speed of 1.8 m/min. On the other hand, the lower the speed, the more evident the conflicting nature of the two objectives: improving the coolant flow settings with respect to one objective makes them worse with respect to the other. In addition, a systematic analysis of the solutions confirms that the actual slab surface temperatures are in most cases higher than the target temperatures, while the core length is shorter than or equal to the target core length.

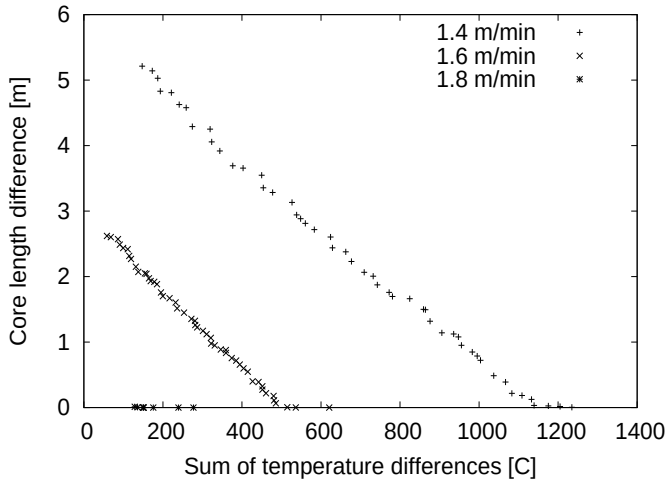

Fig. 1.6 Nondominated fronts of solutions to the two-objective steel casting optimization problem for various casting speeds.

In further experimentation, a detailed analysis of the framework speedup on various numbers of processors was carried out. To make the experimental results directly comparable, the framework parameters other than the number of processors did not vary between the tests. Because the framework is based on the master-slave

parallelization type, the population size was first selected as the one that suits the problem while also being a multiple of the number of processors. As shown in previous work [6, 20], optimization of continuous casting with DE and DEMO seems to work best with population sizes between 20 and 40, which coincides well with the 34 available processors. Number 34 unfortunately has only four divisors (1, 2, 17, and 34). Having numerous divisors is important as it allows for numerous tests where population size is a multiple of the number of processors. Therefore, the population size of 32 was chosen, which has six divisors (1, 2, 4, 8, 16, and 32). With this population size, six tests with various number of processors and maximum efficiency (minimum processor idle time) were possible. In every test, the framework was run five times for each, the single- and multiobjective optimization.

Mean wall clock times of the tests were recorded and are summarized in Table 1.1. Two important observations can be made from the measured wall clock times alone. The first one is great variance of the results. The most likely cause of this is the variable fitness evaluation time, but we will explore this later. To simplify the matters, we will only use mean values of the tests in further discussion. The second observation is that the multiobjective optimization appears slightly slower than the single-objective optimization. The single-sided paired t-test however returns the p value of 0.12, which means the difference in times is not statistically significant. Therefore, both algorithms can be considered equally fast and the following analysis can be generalized in terms of the algorithm choice. Multiobjective optimization will serve as the basis for all further speedup analyses with its differences towards single-objective optimization mentioned only when necessary.

Table 1.1 Mean wall clock times and their standard deviations for the tests with variable number of processors. All times are specified in seconds.

Number of	DE		DEMO	
processors	mean	st. dev.	mean	st. dev.
1	295735	1180	298502	1576
2	143661	945	145584	5646
4	79565	1018	79751	446
8	41412	370	41105	389
16	21123	93	21454	183
32	10925	122	11019	276

We can calculate the speedup directly from the mean wall clock times of the DEMO tests, but let us first try to estimate it with Eq. (1.11). First we make a series of 100 test runs of fitness evaluations from which we estimate the fitness evaluation time to be distributed normally with $\mu = 32.2$ s and $\sigma = 1.5$ s. We estimate all other times in the equation to be in the order of milliseconds and therefore negligible compared to the fitness evaluation time. Now we can estimate the speedup for arbitrary number of processors, N_p, and compare it to the measured speedups. Fig. 1.7 shows the estimated and measured speedups, and the theoretical limit for the speedup on $N_p \in [1..34]$.

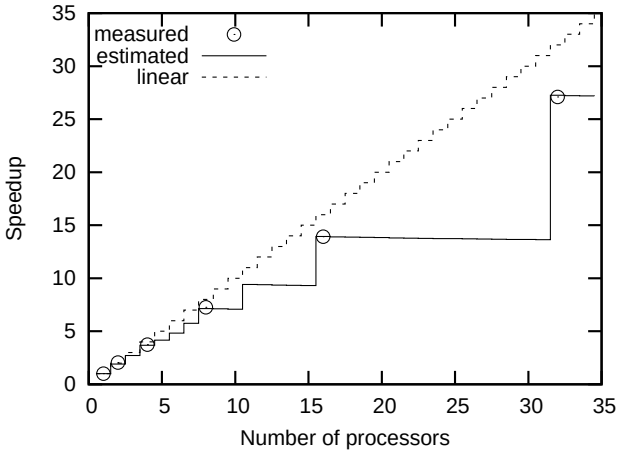

Fig. 1.7 Measured and analytically derived speedup for DEMO on the continuous casting problem, with population size 32, for various number of processors. Linear speedup as the theoretical limit of speedup for master-slave EAs is also shown for reference.

In addition to the total execution time, times of four mutually exclusive steps of the optimization procedure are measured. The first step, which should also be the most time-consuming, is fitness evaluation. The second step is the interprocess communication. This consists of sending the data on individuals from the master process to the slave processes, and sending the fitness evaluation results in the opposite direction. Waiting of the master process for the slave processes to start sending their results is also included in the communication, because in the source code the two are not separated. Next are the output operations, which consist of log keeping and storing the data on the individuals from each generation in a file. The last step is the application of the algorithm operators. The distribution of times among the steps described above for multiobjective optimization on 32 processors is shown in Table 1.2

Table 1.2 The distribution of total wall clock time among steps of the optimization procedure. All times are specified in seconds.

Algorithm stage	mean	st. dev.
total	11019	308
evaluation	9911	376
communication	1108	129
input/output	0.307	0.001
EA operators	0.135	0.003

A quick scan over the times used by the algorithm steps reveals that the algorithm behaves as predicted. Fitness evaluation represents by far the largest part of execu-

tion time, while the times of input/output operations and the evolutionary algorithm operators are negligible in comparison. On the other hand, the interprocess communication time, which should be negligible, represents a substantial proportion of the total algorithm wall clock time. But this view is misleading because the communication times are bundled together with the times of waiting for communication. The latter are a consequence of, and in Eq. (1.11) also a part of, varying fitness evaluation times. We can understand communication times better by analyzing them per generation.

Out of the four steps for which the times are recorded per generation, two – interprocess communication and fitness evaluation – are worth special attention. In addition to the interprocess communication time and the fitness evaluation time as measured on the master process, Fig. 1.8 also shows the maximum time of all fitness evaluations in a generation. It can be seen that the measured communication time roughly equals the difference between the longest fitness evaluation time and the fitness evaluation time on the master process. Measured communication time is therefore mostly spent waiting for the longest fitness evaluations. Pure communication time can be estimated as the sum of communication and the fitness evaluation times on the master process, from which the longest fitness evaluation time is subtracted. It sums up to 1.2 seconds for the shown optimization run, which can be translated to 4 milliseconds per generation on average. Although this is only a rough estimate, it shows that communication times are an order of magnitude longer than the times of the input/output operations and the evolutionary algorithm operators, but still negligible in comparison to the fitness evaluation time. In conclusion, the measured interprocess communication times are in good accordance with the estimates made before the experiments.

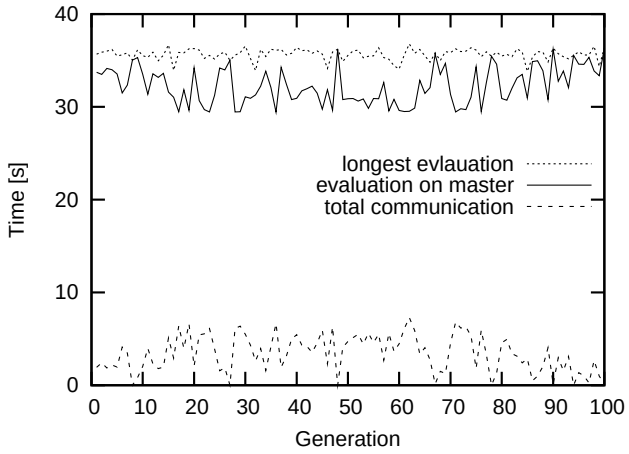

Fig. 1.8 Fitness evaluation and interprocess communication times per generation of multiobjective optimization on 32 processors, for the initial 100 generations. Fitness evaluation time of the master process is contrasted with the longest fitness evaluation time of all the processes.

1.8 Conclusion

In this chapter a parallel evolutionary computation framework for solving numerical optimization problems with one or more objectives was presented. Master-slave parallel versions of the Differential Evolution (DE) and Differential Evolution for Multiobjective Optimization (DEMO) algorithms were implemented for solving single- and multiobjective problems, respectively. The implementation was a straight-forward one, parallelization was done only on the inherently parallel portion of the algorithms – the fitness evaluation, thus keeping the algorithm behavior independent of the number of processors. The interprocess communication was implemented in a simple manner, focusing on its robustness rather than speed.

The performance of the developed framework was empirically evaluated on an industrial optimization problem of tuning coolant flows in the continuous steel casting process. A single- and a two-objective fitness evaluation functions were derived from a computer simulator, implementing a test case of the continuous casting procedure. The quality of the results and the achieved parallel speedups were evaluated separately. The results proved satisfactory and comparable to the results obtained previously on the same problem instances. The measured speedups were high (for example, the speedup on 32 processors was 27) and matched the predictions.

The presented framework demonstrated that due to a relatively simple master-slave parallelization model, EAs can be extensively used on homogeneous parallel hardware. At the same time, it highlighted a weakness of the master-slave model – the sensitivity of the speedup to constant fitness evaluation time. In our case, we experienced variability in the execution time of fitness evaluation at the order of several percent. Similar effect would be expected from a constant-time fitness evaluation function executing on heterogeneous processors or even on homogeneous processors under some load, i.e., executing other jobs. Therefore, our future work will focus on overcoming the demand for constant fitness evaluation time. This will be achieved by eliminating the synchronous nature of the master-slave parallelization type and thus maximizing the algorithm efficiency (minimizing processor idle time). In this way we expect to increase the speedup and make the algorithms more usable on heterogeneous hardware architectures that are less suitable to ordinary master-slave EAs.

Acknowledgements The authors are grateful to Professor Erkki Laitinen from the Department of Mathematical Sciences, University of Oulu, Finland, for providing the mathematical model and technical details of the continuous casting process optimized in this study. The work was supported by the Slovenian Research Agency under research programmes P2-0095 *Parallel and Distributed Systems*, and P2-0209 *Artificial Intelligence and Intelligent Systems*.

References

1. A. E. Eiben, J. E. Smith, Introduction to Evolutionary Computing, Springer-Verlag, Berlin, 2003.
2. K. De Jong, Evolutionary Computation: A Unified Approach, The MIT Press, Cambridge, 2006.
3. G. B. Fogel, D. W. Corne (Eds.), Evolutionary Computation in Bioinformatics, Morgan Kaufmann Publishers, Amsterdam, 2003.
4. D. Dasgupta, Z. Michalewicz (Eds.), Evolutionary Algorithms in Engineering Applications, Springer-Verlag, Berlin, 1997.
5. J. Biethahn, V. Nissen (Eds.), Evolutionary Algorithms in Management Applications, Springer-Verlag, Berlin, 1995.
6. B. Filipič, E. Laitinen, Model-based tuning of process parameters for steady-state steel casting, Informatica 29 (4) (2005) 491–496.
7. R. Buyya, High Performance Cluster Computing: Architectures and Systems, Vol. 1, Prentice Hall, Upper Saddle River, 1999.
8. K. V. Price, R. Storn, Differential evolution: A simple evolution strategy for fast optimization, Dr. Dobb's Journal 22 (4) (1997) 18–24.
9. K. Price, R. M. Storn, J. A. Lampinen, Differential Evolution: A Practical Approach to Global Optimization (Natural Computing Series), Springer-Verlag, Berlin, 2005.
10. K. Deb, Multi-Objective Optimization using Evolutionary Algorithms, John Wiley & Sons, Chichester, UK, 2001.
11. C. A. Coello Coello, D. A. Van Veldhuizen, G. B. Lamont, Evolutionary Algorithms for Solving Multi-Objective Problems, Kluwer Academic Publishers, New York, 2002.
12. A. Abraham, L. Jain, R. Goldberg (Eds.), Evolutionary Multiobjective Optimization, Springer-Verlag, London, 2005.
13. T. Robič, B. Filipič, Demo: Differential evolution for multiobjective optimization, in: C. A. Coello Coello, A. Hernández Aguirre, E. Zitzler (Eds.), Conference on Evolutionary Multi-Criterion Optimization, Vol. 3410 of Lecture Notes in Computer Science, Springer, Berlin, 2005, pp. 520–533.
14. K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, IEEE Transactions on Evolutionary Computation 6 (2) (2002) 182–197.
15. S. G. Akl, Parallel Computation: Models and Methods, Prentice Hall, Upper Saddle River, 1997.
16. E. Cantú-Paz, A survey of parallel genetic algorithms, Tech. rep., University of Illinois at Urbana-Champaign (1997).
17. D. A. van Veldhuizen, J. B. Zydallis, G. B. Lamont, Considerations in engineering parallel multiobjective evolutionary algorithms, IEEE Transactions on Evolutionary Computation 7 (2) (2003) 144–173.
18. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra, MPI – The Complete Reference, The MIT Press, Cambridge, 1996.
19. A. J. Nebro, F. Luna, E.-G. Talbi, E. Alba, Parallel multiobjective optimization, in: E. Alba (Ed.), Parallel Metaheuristics, John Wiley & Sons, New Jersey, 2005, pp. 371–394.
20. B. Filipič, T. Tušar, E. Laitinen, Preliminary numerical experiments in multiobjective optimization of a metallurgical production process, Informatica 31 (2) (2007) 233–240.

Index