Asynchronous Master-Slave Parallelization of Differential Evolution for Multiobjective Optimization

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

Roman Trobec roman.trobec@ijs.si
Department of Communication Systems, Jožef Stefan Institute, Jamova cesta 39, SI-1000, Ljubljana, Slovenia

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

Abstract

In this paper, we present AMS-DEMO, an asynchronous master-slave implementation of DEMO, an evolutionary algorithm for multiobjective optimization. AMS-DEMO was designed for solving time-demanding problems efficiently on both homogeneous and heterogeneous parallel computer architectures. The algorithm is used as a test case for the asynchronous master-slave parallelization of multiobjective optimization that has not yet been thoroughly investigated. Selection lag is identified as the key property of the parallelization method, which explains how its behavior depends on the type of computer architecture and the number of processors. It is arrived at analytically and from the empirical results. AMS-DEMO is tested on a benchmark problem and a time-demanding industrial optimization problem, on homogeneous and heterogeneous parallel setups, providing performance results for the algorithm and an insight into the parallelization method. A comparison is also performed between AMS-DEMO and generational master-slave DEMO to demonstrate how the asynchronous parallelization method enhances the algorithm and what benefits it brings compared to the synchronous method.

Keywords

multiobjective optimization, evolutionary algorithms, differential evolution, parallelization, distributed computing, speedup, selection lag.

1 Introduction

Real-world optimization problems are often high-dimensional, requiring the use of stochastic optimization techniques, such as evolutionary algorithms (EAs). Multiobjective optimization (MO) is the process of simultaneous optimization of two or more conflicting objectives, resulting in a set of solutions that represent various tradeoffs between the objectives. As population-based methods, EAs can be extended to return multiple solutions in a single run, which makes them suitable for multiobjective optimization.

In addition, real-world problems are frequently time demanding, requiring the use of parallel computer architectures for the optimization to have any practical value. Before parallelizing an EA, properties of the parallel computer architecture should be considered, since the selection of the most appropriate parallelization method depends on them. The most important is the distinction between homogeneous and heterogeneous computer architectures; the components of the latter differ in their processing power as well as communication speed, making them more difficult to use efficiently and narrowing the choice of applicable parallelization methods. The goal of this paper is to present the AMS-DEMO algorithm, a parallel evolutionary algorithm for solving time-demanding multiobjective optimization problems, that is able to utilize various parallel computer architectures, both homogeneous and heterogeneous, regardless of the properties of interconnections, and with no limits on the number of processors. The algorithm is parallelized using an asynchronous master-slave method. Although similar asynchronous master-slave methods have been used for parallelization of EAs (Stanley and Mudge, 1995; Talbi and Meunier, 2006) and particle swarm optimization (Kennedy and Eberhart, 1995), and displayed good performance on heterogeneous computer architectures, no analysis of their asynchronism exists to our knowledge. This is possibly due to EAs being experimentally evaluated, paired with the difficulty of generalizing the experiments performed on heterogeneous computer architectures. We combine the experimental evaluation of the algorithm on a homogeneous computer architecture and heterogeneous grid architecture with a theoretical analysis of the algorithm behavior that provides an insight into the parallelization method.

This paper is further organized as follows. Section 2 presents the background of multiobjective optimization evolutionary algorithms, their representative used in this paper – the Differential Evolution for Multiobjective Optimization (DEMO) – and a review of the possible methods of parallelization. Section 3 presents the main contribution of the paper, on both the conceptual and the implementation level, with all the important details explained. Experiments with the proposed algorithm on a benchmark problem and a real-world industrial problem follow in Section 4. The paper concludes with Section 5, where possible limitations of the algorithm and future work are discussed.

2 Background

2.1 Multiobjective optimization

Multiobjective optimization problems are tasks that require optimizing (finding either minimum or maximum of) a vector function:

$$\mathbf{y} = \mathbf{f}(\mathbf{x})$$

where $\mathbf{x}$ is a vector of n decision variables defined over $\mathbb{R}$

$$\mathbf{x} = (x_1, x_2, \ldots, x_n)$$

and $\mathbf{y}$ is a vector of m objectives

$$\mathbf{y} = (y_1, y_2, \ldots, y_m).$$

Decision variable vectors $\mathbf{x}$ that satisfy inequality constraints

$$g_i(\mathbf{x}) \geq 0,\ i = 1, 2, \ldots, I$$

and equality constraints

$$h_i(\mathbf{x}) = 0,\ j = 1, 2, \ldots, J$$

are called feasible solutions. There are two Euclidean spaces associated with multiobjective optimization. These are the n-dimensional *decision variable space* of solutions to the problem, and the m-dimensional *objective space* of their images under $\mathbf{f}$. The latter is partially ordered according to the *Pareto dominance* relation. Given two objective vectors, $\mathbf{a}$ and $\mathbf{b}$; $\mathbf{a}$ is said to *dominate* $\mathbf{b}$ iff $\mathbf{a}$ is not worse than $\mathbf{b}$ in all objectives and is better than $\mathbf{b}$ in at least one objective. Formally, assuming a minimization problem:

$$\begin{aligned} \mathbf{a} \prec \mathbf{b} \quad & \text{iff} \\ \forall k \in \{1, 2, \ldots, m\} : a_k \leq b_k \quad & \text{and} \\ \exists l \in \{1, 2, \ldots, m\} : a_l < b_l \quad & \end{aligned} \tag{1}$$

The solution to a multiobjective optimization problem, called the Pareto optimal set, is a set of feasible solutions in the decision variable space, whose images in the objective space, called the Pareto front, are not comparable with each other and are not dominated by any feasible solution. The Pareto optimal front forms a hypersurface in the objective space. The task of multiobjective optimization is to find a nondominated set of solutions, representing an approximation for the Pareto front. This is a preparatory step to assist the user in deciding on the final solution, using additional preferences.

2.2 DE and DEMO

DE is an evolutionary algorithm for solving single-objective optimization problems defined over continuous domains (Price and Storn, 1997; Storn and Price, 1997; Price et al., 2005). Variation operators of DE are differential mutation and uniform crossover. In every iteration, an individual, called a parent, is selected from the population at random, but in such a way that every population member acts as a parent in one generation. Differential mutation takes three or more members of the population $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, [\mathbf{x}_4 \ldots \mathbf{x}_m] \in \mathbb{P}$, to help construct a mutation vector $\mathbf{v}$ by vector addition and scalar multiplication. A simple way of calculating a mutation vector is by $\mathbf{v} = \mathbf{x}_1 + F \cdot (\mathbf{x}_2 - \mathbf{x}_3)$, where $F \in \mathbb{R}$ and is often from the interval $(0, 1]$. The mutation vector is then recombined with the parent by uniform crossover, creating a trial solution. The trial solution and the parent then undergo selection, after which the better of the two is selected for the next generation and the other discarded.

Robič and Filipič (2005) presented DEMO, which was devised from DE by implementation of the following changes:

- The algorithm was changed from generational to steady-state. Unlike generational algorithms, steady-state algorithms (Eiben and Smith, 2003) immediately evaluate the solutions they create – most often one or two new solutions are created by variation operators – and replace their parents in the population, before variation operators are applied again to the population. This change is straightforward to implement in DEMO – after each selection, the surviving solutions (the trial or the parent or both) immediately replace the parent solution in the current population instead of placing them into a new population.

- When the parent is compared to the trial solution, the Pareto dominance is used, resulting in three possible outcomes. The first two are that one solution dominates the other, and is thus used for the next generation, while the other is discarded,

like in DE. The third option is that no solution dominates the other, and in this case both solutions are kept, increasing the population size by one.

- A mechanism was added to counterbalance the increases in population size. Every n evaluations, where n is the starting population size, the population is truncated using the nondominated sorting and the crowding distance metric known from the NSGA-II algorithm by Deb et al. (2002).

DEMO has been extensively tested by Robič and Filipič (2005) on ZDT test problems (Zitzler et al., 2000); and compared by Tušar and Filipič (2007) with IBEA (Zitzler and Künzli, 2004), SPEA-2 (Zitzler et al., 2001), and NSGA-II on DTLZ test problems (Deb et al., 2005) and WFG test problems (Huband et al., 2005). DEMO was found to outperform other algorithms on a large subset of these problems. A DEMO variant with a mechanism for self-adaptation of DE parameters, DEMOwSA by Zamuda et al. (2007), participated in CEC 2007 Competition on Performance Assessment of Multi-Objective Optimization Algorithms (Suganthan, 2007). Based on the previous extensive comparisons, DEMO behavior is well known, and DEMO was therefore selected as the base algorithm for the parallelization.

2.3 Parallelization of EAs

In solving real-life optimization problems, fitness evaluation is sometimes very computationally expensive as, for example in microprocessor design (Stanley and Mudge, 1995) and airplane wing design (Quagliarella and Vicini, 1998; Sasaki et al., 2001). It is therefore beneficial to parallelize the algorithm for use on multiple processors and thus shorten the execution time. A two-level hierarchical parallelization first parallelizes fitness evaluation on p_1 processors, and then parallelizes optimization on p_2 groups of processors. A total of p_1 * p_2 processors can be used in this kind of highly efficient parallelization. The problems of parallelizing fitness evaluation and optimization are disjunct, and since the options for efficient parallelization of fitness evaluation are problem dependent, we only work on parallelization of the optimization algorithm in this paper.

There are several categorizations of parallel metaheuristics for multiobjective optimization (Nebro et al., 2005; Talbi et al., 2008). Here, we present one that is most commonly used when dealing with EAs. There are four EA parallelization methods (Cantú-Paz, 1997; Alba and Troya, 2002; van Veldhuizen et al., 2003; Luna et al., 2006) – three basic, i.e. the *master-slave* (also called the *global parallelization*), the *island model*, and the *diffusion model* (also known as the *cellular model*), and the *hybrid model* that encompasses combinations of the basic types, usually in a hierarchical structure.

Master-slave EAs are the most straightforward type of parallel EAs because they build on the inherent parallelism of EAs. Consequently, they traverse the search space identically to their serial counterparts. They can be visualized as a master node running a serial EA, but with a modification in creation and evaluation of solutions, which happen on all available processors in parallel. This, however, does not apply to steady-state algorithms, in which the creation and evaluation of a single solution depend on the previously generated solution and the result of its evaluation. Steady-state algorithms can therefore not be parallelized using the master-slave type without prior modification.

The highest efficiency of the master-slave parallelization type can be achieved on computers with homogeneous processors and in problem domains where the fitness evaluation time is long, constant, and independent of the solution. When these criteria are fulfilled, near-linear speedup (Akl, 1997) (speedup that is close to the upper

theoretical limit) is possible. The master-slave parallelization is popular with MOEAs, ranging from simple implementations as in the case of Oliveira et al. (2003) where the master runs on a separate processor, and the cases of Radtke et al. (2003) and Nebro and Durillo (2010) where the master node also runs one slave.

There are also implementations for heterogeneous computer architectures where load-balancing has to be implemented. Examples can be found in Eberhard et al. (2003) with pool-of-tasks load balancing algorithm, Lim et al. (2007) where a grid-enabled algorithm combines the island model with the master-slave model, Stanley and Mudge (1995), and Talbi and Meunier (2006) with an asynchronous master-slave parallelization of a steady-state algorithm where load balancing is implicit. Similar methods have been used to parallelize particle swarm optimization (PSO) methods with good results. Asynchronous master-slave model is compared to synchronous on various numbers of processors and varying evaluation time in Koh et al. (2006). Multiobjective PSOs were parallelized in Scriven et al. (2008), Mostaghim et al. (2008), and Lewis et al. (2009) with mixed asynchronous and synchronous master-slave model, resulting in an algorithm that works well on unreliable heterogeneous computer systems. Yet the connection between the number of processors and the behavior of any of the asynchronous master-slave algorithms has not been analyzed.

3 The AMS-DEMO algorithm

3.1 Parallelizing DEMO

By means of parallelization, the proposed asynchronous master-slave DEMO (AMS-DEMO) attempts to speedup solving real-world optimization problems with expensive evaluation functions. It is designed to be used on problems where evaluation of solutions, including constraint checks, takes several orders of magnitude longer than other parts of the algorithm, and to run efficiently on both homogeneous and heterogeneous computer architectures. Because it is not designed to be efficient on computationally inexpensive problems, its use in combination with surrogate models is not envisioned. The parallelization seeks to modify the algorithm in a way that maximizes speedup (Akl, 1997) – the factor of how much shorter the execution time is on multiple processors than on a single processor.

We used DEMO as a base algorithm (more specifically: the variant DEMO/parent described in Robič and Filipič (2005), using DE/rand/1/bin scheme) because it already proved successful in solving numerical multiobjective optimization problems (Tušar and Filipič, 2007), outperforming other state-of-the-art algorithms. Its application in the optimization of a real-world steel casting process (Filipič et al., 2007), however, proved very time consuming, with a single run taking more than 3 days to complete on an average PC. To speedup the DEMO algorithm, it was parallelized to run on a computer cluster. Given the properties of the available cluster (32 identical processors with fast interconnections) and problem properties (solution evaluation time is slightly variable but input independent and orders of magnitude longer than variation operators of DEMO), the master-slave parallelization model was selected as the most appropriate.

Initially, the DEMO algorithm was transformed into a generational algorithm and parallelized with the standard master-slave model in Filipič and Depolli (2009), because it offers reasonable speedups while being easy to implement. We call this algorithm generational DEMO and use it as a baseline for comparison in Section 4 of this paper. Once every generation, generational DEMO synchronizes all of the processors. This happens when the master calculates the population of the next generation while the slaves wait. In the best case scenario, all the slaves finish at the same time and only wait

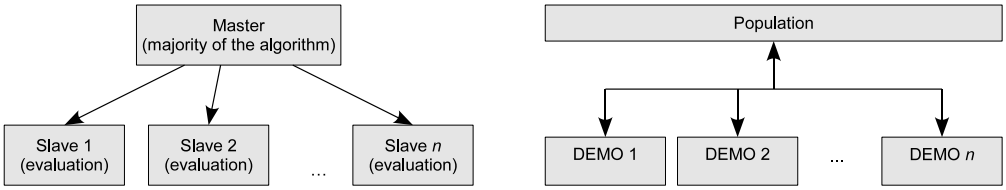

Figure 1: Schematic representation of AMS-DEMO on two levels. On the process level (left), AMS-DEMO follows the standard master-slave division of operations between the master and the slaves. On the conceptual level (right) it can be viewed as a number of original DEMO algorithms sharing a single population, but being otherwise independent, which differs greatly from the standard master-slave algorithms that are equivalent to a serial algorithm.

for the master to calculate and distribute the next generation – a task that is negligible in duration compared to a solution evaluation. This is, however, only possible if all the slaves are load balanced, which in practice translates to all the slaves being equally fast, performing the same number of evaluations per generation – requiring the population size to be a multiple of the number of processors, and all evaluations being equally time demanding. Although these conditions are often met to a high degree, greater flexibility is desired and as we show next, can be achieved for the cost of a minor drop in the algorithm convergence rate.

3.2 Asynchronous master-slave

By shifting the main task of the algorithm from traversing the search space in the same way as on a single processor, towards keeping the slaves constantly occupied, we created AMS-DEMO that greatly exceeds the flexibility of generational DEMO. The master-slave model is used for parallelization, with the slaves running on all the p processors and the master running as an additional process on one of the processors. Conceptually, AMS-DEMO operates as p asynchronous and independent DEMO algorithms processing a shared population, as shown on the right-hand side of Figure 1. The population is stored on the master, where the variation operators and selection are also applied, while the slaves only evaluate solutions supplied to them by the master, as seen on the left-hand side of Figure 1. Input parameter bounds are checked and enforced on the master while all problem dependent constraint checks are performed on the slaves, within the evaluation (evaluation time includes all the constraint checks). Because the slaves are asynchronous, there is no need to explicitly load-balance them. In practice this means AMS-DEMO is efficient in using heterogeneous computer architectures, computers with varying background load, dynamic numbers of processors, and there are no performance based restrictions on the population size or the number of processors.

The slaves only wait a minimum amount of time between evaluations, while the master performs operations orders of magnitude shorter, and spends most of the time waiting. This, however, does not decrease the efficiency of the algorithm because the master shares a processor with one of the slaves, and both processes are implemented with non-blocking wait. The shared processor is thus either executing the master or the slave and is never idle.

The communication between the master and the slaves is in the form of asynchronous message passing. Message passing means that communication consists of

a sender sending a message and a receiver receiving the message. The asynchronous nature of the communication is manifested as the ability of the sender and the receiver to handle messages independently of each other. In contrast, the synchronous message passing used in generational DEMO requires the sender and the receiver to participate in the communication simultaneously, in effect synchronizing them. Generational DEMO requires the processors to be synchronized at the time the messages are either gathered from the slaves or sent to the slaves, making the synchronous message passing perfect for the task. For AMS-DEMO, the asynchronous message passing is an advantage because it requires no unnecessary synchronization or the wait times associated with it.

To use asynchronous communication to its full extent, AMS-DEMO introduces FIFO (first in, first out) queues of solutions pending evaluation on the slaves. A slave with a local queue is able to start evaluation of a solution from the front of its queue immediately after it completes its previous evaluation. It only briefly interrupts the chain of continuous evaluations by sending the last evaluation results to the master, and by checking for and processing any pending messages from the master. Both of these operations are fast because of the asynchronous communication.

Note that the queue of length one is equivalent to no queue at all, because the solution at the front of the queue is the one being evaluated – a slave only removes it after it has evaluated it. In most cases, a queue of length two should suffice to eliminate all the wait time for the slave, because the slave does not require more than one solution waiting in the queue. There are exceptions, such as the cases where the communication time is of the same order of magnitude as the evaluation time, and the cases where it is beneficial to send more than one solution per message because of expensive communication.

3.3 Selection lag

We explore an important difference between AMS-DEMO and the original DEMO – the difference in the way solutions are related to the population. The difference can easily be demonstrated if we observe a typical solution from its creation to its selection. While in the original DEMO, the population does not change in this observed time period, it may change in AMS-DEMO. The change happens because, in AMS-DEMO, while the observed solution is being evaluated on one processor, some number of other solutions complete their evaluation on other processors, and are sent to the master, undergo selection, and may thus change the population. This causes a lag in exploitation of good solutions. We will call it *selection lag*, denote it with l, and define it, per solution, as the number of solutions that undergo selection in the time between the observed solution's creation and selection. The selection lag therefore counts the number of possible changes to the population (the number of replaced solutions) that are not known to AMS-DEMO when it creates the observed solution, but would be known to the original DEMO. Because every selection is coupled with the creation of a new solution, the selection lag can also be thought of as the number of solutions created while an observed solution is being evaluated – in other words, as the number of solutions that could be created differently in the original DEMO than they are in AMS-DEMO, because of the different processing of the observed solution. It should be stressed that the changes to the population counted by the selection lag are possible, but not necessary. Furthermore, although their probabilities depend linearly on the selection lag, they also depend on the population size and on the probability of the offspring surviving selection.

Defined per solution, in a single run of AMS-DEMO, the selection lag becomes a statistical variable which characterizes the behavior of AMS-DEMO. We can assume, however, that as long as the selection lags of solutions deviate little from the mean, the errors made by observing only the mean selection lag, which is easily calculated as $\bar{l} = pq - 1$, are negligible.

The proof for $\bar{l} = pq - 1$ is given through the following example. In the simplest case, with queue size 1 and equal evaluation times, slaves work as follows. They are assigned solutions and start evaluating them in an orderly fashion. Slave 1 is assigned solution 1, slave 2 is then assigned solution 2 and so on until the last slave p is assigned solution p. Equally fast slaves finish evaluations and receive the next p solutions in the same order as the first p solutions. Slave 1 evaluates solution 1 and is assigned solution $p + 1$. Therefore, the selection lag for solution 1, given as the number of solutions created during its evaluation, equals $p - 1$, as does for all other solutions. The mean selection lag is then also $p - 1$. In a more realistic case, where the evaluation times vary, the selection lag may no longer equal to $p - 1$ for all solutions. Any increase in one solution's selection lag, however, must produce an equivalent decrease in selection lags of other solutions. This can be demonstrated with an example of two solutions, a and b, evaluated in parallel, with a undergoing selection just prior to b. If the evaluation time of a were increased just enough for it to undergo selection just after b, its selection lag l_a would increase by 1. But this would automatically decrease the selection lag of b by 1, because a would no longer undergo selection, while b was being evaluated. Thus any transposition of the evaluation order of two solutions changes their selection lags symmetrically, so that their mean does not change. This rule can also be extended to all possible permutations of the evaluation order, since any permutation can be represented as a composition of transpositions. With the introduction of queues, the time between creation and selection of a solution lengthens by the time the solution waits in the queue. The number of solutions generated in a certain period of time equals the number of solutions already in the queue, $q - 1$, plus the number of solutions generated on the other processors, $(q - 1)(p - 1)$. The selection lag of an average solution is the total number of solutions generated while this solution is waiting or being evaluated: $(q - 1) + (q - 1)(p - 1) + (p - 1)$, which simplifies to $pq - 1$.

3.4 Implementation details

The AMS-DEMO algorithm consists of the master and slave processes. Both communicate with each other using the Message Passing Interface (MPI) (Snir et al., 1996) communication standard. The algorithm however does not depend on this standard, so any asynchronous communication protocol should be adequate.

The AMS-DEMO slave process is summarized in the pseudo code as follows:

AMS-DEMO – slave process

```
1: create empty FIFO queue Q
2: while termination not requested do
3:   if Q empty then
4:     if no pending messages from the master then
5:       wait for a message
6:     end if
7:     push solutions from received messages into Q
8:   else
9:     evaluate the first element from Q
```

```
      send the evaluation results to the master
      remove the first element from Q
    end if
end while
```

Although the AMS-DEMO master process mainly implements the functionality of the original DEMO, the inclusion of communication and slave supervision complicates it somewhat. The following pseudo code provides the details:

AMS-DEMO – master process

```
create p empty queues Q_1 ... Q_p to serve as copies of the queues on the slaves
create empty population ℙ
parent_index ← 1
num_evaluated ← 0
while stopping criterion not met do
  while ∃k : |Q_k| ≤ queue_size do
    c ← Create(parent_index)
    select index j such that ∀k ∈ [1 ... n] : |Q_j| ≤ |Q_k|
    add c to Q_j
    send a message containing c to slave S_j
    parent_index ← (parent_index mod n) + 1
  end while
  wait for messages from the slaves
  while pending messages do
    extract solution c from the first pending message
    j ← the number of slave that sent the message
    remove c from Q_j
    Selection(c)
    num_evaluated ← num_evaluated + 1
    if num_evaluated is a multiple of n then
      if |ℙ| > n then
        truncate ℙ
      end if
      randomly enumerate solutions from ℙ
    end if
  end while
end while
send termination request to all slaves
```

While the original DEMO provides an initialization step, in which the initial population is generated and evaluated, AMS-DEMO does not, thus avoiding the required synchronization of processes associated with such a step. AMS-DEMO rather starts with an empty population $\mathbb{P}$ and modifies the way it creates solutions in its main loop, as can be seen from the function Create:

Create(i)

```
if |ℙ| < n then
  randomly create a solution c
  mark c as unevaluated
  mark c as the parent of itself
  append c to ℙ
```

```
6: else
7:    randomly select three different solutions x_1, x_2, x_3 from P
8:    create a candidate solution c ← x_1+F·(x_2 − x_3)
9:    select p_i, the i-th element of P
10:   alter c by crossover with p_i
11:   mark p_i as the parent of c
12: end if
13: return c
```

Because the solutions of the initial population are now created in the main loop, selection must be able to detect them and allow them to bypass it. This is done in function Selection by testing the solution's parent:

Selection(c)

```
1: locate parent p of c in P
2: if p not found in P then
3:    select p as a random element of P
4: end if
5: if p is not evaluated then
6:    replace p with c
7: else
8:    compare c to p
9:    if c dominates p then
10:      replace p with c
11:   else if p dominates c then
12:      keep p
13:   else
14:      keep p and add c to P
15:   end if
16: end if
```

If the parent is marked as unevaluated, then the offspring replaces it, bypassing the selection. There are two possible scenarios resulting in the parent being marked as unevaluated. The first, more common one, is that the solution is from the initial population, having no real parent, and is marked as the parent of itself (line 4 of function Create). In the second, rarer scenario, the parent is a member of the initial population and has not yet been evaluated. In such a case, the two related solutions (the parent and the offspring) simply switch their roles. The offspring skips the selection and replaces the parent in the population. Then, after the parent is evaluated, the parent undergoes selection in which it competes against the offspring. Because of the asynchronous nature of AMS-DEMO, the parent of an observed solution might not be found in the population, because it was already replaced by some other solution or eliminated by the population truncation. When this happens, a random solution from the population is selected as the parent to compete against the observed solution.

4 Algorithm evaluation

The performance of the proposed algorithm was assessed on a benchmark problem and a real-world multiobjective optimization problem. The benchmark problem is used to discover the relation between the computational complexity of the evaluation function and the efficiency of AMS-DEMO. The real-world problem exercised for testing

the algorithm convergence, parallel speedup, and the ability to run on heterogeneous systems with numbers of processors larger than the population size. Experiments on both problems were performed on a cluster of 16 dual AMD Opteron 244 processor computers, each with 1024 MB of memory, and six gigabit Ethernet ports, connected with a gigabit Ethernet switch. The software used includes Fedora Core 2 Linux with kernel 2.6.8-1.521smp, MPICH 1.2.7. communication library (Gropp et al., 1996) and GCC 3.3.3 compiler.

4.1 Experiments with evaluation time

Because of its master-slave parallelization, the AMS-DEMO master process has to communicate twice with the slaves for every individual it creates. This limits the AMS-DEMO usability on problems with computationally inexpensive evaluation functions. If evaluation time is comparable to communication time, then there is no gain in sending solutions into evaluation on other processors, since the parallel algorithm would require a comparable amount of time communicating as the sequential algorithm for the whole execution. Therefore, in the first set of experiments we explore how long the evaluation of a solution should last for AMS-DEMO to be able to evaluate the same number of solutions in less time than the original DEMO. For timing we use the Linux high-precision timer function *clock_gettime* with nanosecond resolution and mean response time of 30 ns. We use the SYMPART benchmark function from the CEC 2007 Competition on Performance Assessment of Multi-Objective Optimization Algorithms (Suganthan, 2007) and add a variable length delay to it. Because DEMO has already proved successful in solving this function (Zamuda et al., 2007), we perform no additional experiments regarding the solution quality with AMS-DEMO. We merely use the function to demonstrate how AMS-DEMO performs on problems with extremely inexpensive fitness evaluation functions. We set population size to 100, scaling factor F to 0.5, crossover probability to 0.1, DE scheme to rand/1/bin, and the stopping criterion to 5000 evaluations. We perform tests on $n = \{1, 2, 4, 8, 16, 32\}$ processors and vary the evaluation function time t_e from the set of values $\{5.5\ \mu\text{s}, 10\ \mu\text{s}, 100\ \mu\text{s}, 1\ \text{ms}, 10\ \text{ms}, 100\ \text{ms}, 1\ \text{s}, 10\ \text{s}\}$. The lowest value for t_e is the evaluation time of the SYMPART fitness function with no additional delay on the hardware used in experiments. In each test, we measure AMS-DEMO execution time $t(n)$. We take the definition for *weak speedup* (type II.B speedup as defined in Alba (2002)) $S_e(n)$, as how much faster the parallel algorithm on n processors performs a fixed number of evaluations than the original sequential algorithm, and calculate it as:

$$S_e(n) = \frac{t(1)}{t(n)} \tag{2}$$

We call it weak speedup, because it is based on the execution time of an algorithm with the number of evaluations set as the stopping condition. True speedup, in contrast, would have solution quality as the stopping condition, but that would make it more problem dependent. In this analysis we are interested solely in the response of AMS-DEMO to various evaluation time lengths and therefore we chose the weak speedup over the true speedup to make the results as widely applicable as possible. We develop a simple model for estimating the weak speedup of AMS-DEMO on p processors. We examine the time required for p evaluations and all the algorithm overhead. For simplicity, we intentionally leave out some factors that are hard to determine in advance for any given computer system. Nevertheless, we need to introduce three specific times. t_a is the algorithm overhead time, defined per evaluation. The algorithm

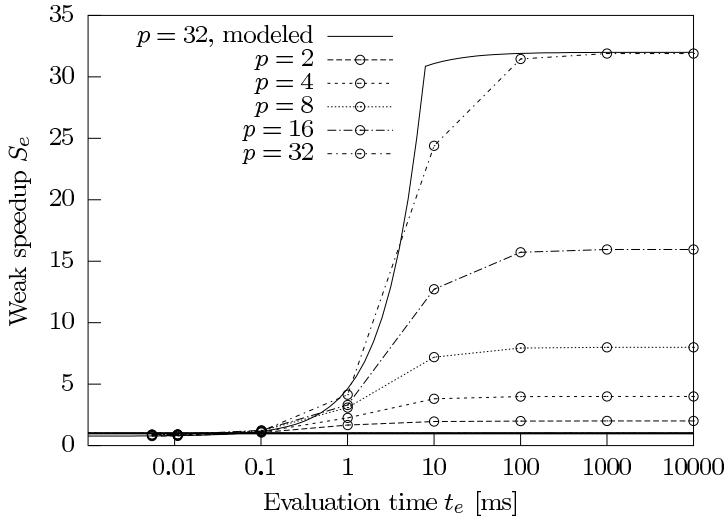

Figure 2: Weak speedup S_e as the function of evaluation time t_e for various p. Also plotted is the modeled S_e for $p = 32$ that matches the measured values well in all points but $t_e = 10$ ms. $S_e = 1$ is drawn with a thick line for reference.

overhead includes all algorithm tasks but evaluation, e.g., application of variation operators and selection, output of solutions and evaluation results to a file, etc. t_n is the network latency – the time it takes for one processor to fully transfer a message to a remote processor, or to fully receive a message from the remote processor. During the network latency, the processor may perform other tasks, because all of the work is done by either networking hardware or the remote processor. There is however an overhead for the processor, associated with the communication – the time required to prepare data and pack it into a message or to unpack the data from the message and copy it to local variables (depending at which side of communication the said processor is). We refer to this time as communication overhead time, t_c.

Our model calculates the weak speedup as the ratio of the time DEMO performs p evaluations and the time AMS-DEMO on p processors performs the same number of evaluations. DEMO has no communication overhead and performs all its tasks sequentially, therefore it completes p evaluations in time $pt_e + pt_a$. AMS-DEMO must perform all tasks that deal with any particular slave sequentially, that is, communicate with the slave (send a solution), wait for the slave to evaluate the solution, communicate with the slave again (receive the evaluation results), and then perform one set of algorithm overhead tasks. With the exception of communication overhead, communicating with one slave can overlap with all other tasks associated with the other slaves, because the network hardware works independently of the processor. Because of the overlap, AMS-DEMO on p processors performs p evaluations in time $\max(pt_a + 2pt_c,\ 2t_c + 2t_n + t_e)$. For simplicity, we ignore that the master process must execute on the same processor as one slave process. Our model can be thus described as:

$$S_e \approx \frac{pt_e + pt_a}{\max(pt_a + 2pt_c,\ 2t_c + 2t_n + t_e)} \tag{3}$$

In Figure 2 we plot the the experimentally measured S_e (mean of 25 runs) for vari-

ous p. Interestingly, weak speedups are less than 1 for $t_e < 0.1$ and all p. We can see that for $t_e \geq 0.1$ ms, AMS-DEMO weak speedup rises above 1. At $t_e = 10$ ms, AMS-DEMO already reaches nearly full speedup ($S_e \approx n$) at low numbers of processors ($n \leq 4$). For higher numbers of processors, evaluation time must be even higher for AMS-DEMO to achieve full speedup. We also modeled and plotted S_e for 32 processors in the same figure (labeled as $p = 32$, *modeled*) using values for $t_n = 0.22$ ms, $t_c = 0.03$ ms, and $t_a = 0.20$ ms, measured during the experiments. The simple model is accurate on most of the values for t_e, and only fails to predict the weak speedup at $t_e = 10$ ms. Even with one inaccurate prediction, the simple model should be useful for deciding on when to use AMS-DEMO.

4.2 The real-life optimization problem

Next, AMS-DEMO was tested on a real-world multiobjective optimization problem of tuning coolant flows in industrial continuous casting of steel. Continuous casting is widely used at steel plants to produce semi-products of various shapes and dimensions. The process starts by pouring liquid steel, melted in a furnace, into the mold, a bottomless vessel cooled by water flow in its walls. Cooling in the mold, also referred to as primary cooling, extracts heat from the steel and initiates the formation of a solid shell on its surface. The shell formation is crucial for the support of the steel slab after it exits the mold and enters into the secondary cooling area where it is cooled by water sprays. Led through the secondary cooling area by support rolls, the slab progressively solidifies and finally exits the casting machine. At its outlet it is cut into pieces of predetermined length.

Cooling of the steel during continuous casting affects the safety and productivity of the casting process, as well as the quality of the cast steel. Tuning of the water spray flows in the secondary cooling zone is, in particular, a demanding task, since the secondary cooling region is divided into a number of cooling zones where the amounts of the cooling water can be set separately. The casting machine considered in our experiments involves a secondary cooling area divided into nine zones. In every zone, the cooling water is dispersed to the slab at the center and corner positions, which results in 18 water flows that need to be appropriately tuned. Based on empirical knowledge, target temperatures are specified for the slab center and corner in each zone. Tuning has to be carried out in such a way that the resulting slab surface temperatures match the target temperatures. This goal is formally defined by an objective function measuring the differences between the actual and target temperatures:

$$f_1 = \sum_{i=1}^{N_Z} |T_i^{\text{center}} - T_i^{\text{center}*}| + \sum_{i=1}^{N_Z} |T_i^{\text{corner}} - T_i^{\text{corner}*}|, \tag{4}$$

where N_Z is the number of zones, T_i^{center} and T_i^{corner} are the slab surface temperatures at the center and the corner positions in zone i, and $T_i^{\text{center}*}$ and $T_i^{\text{corner}*}$ the target temperatures in zone i. This objective function has to be minimized in order for the casting process to result in high-quality steel.

An additional objective in this process refers to the core length, l^{core}, which is the distance from the mold exit to the point of complete solidification of the slab. This is also affected by the coolant flow setting. Its target value, $l^{\text{core}*}$, is prespecified, and the actual core length should be as close to it as possible. Shorter core length may result in unwanted deformation of the slab, while longer core length has to be avoided for

safety reasons. This gives the second objective function to be minimized:

$$f_2 = |l^{\text{core}} - l^{\text{core}*}|. \tag{5}$$

Water flows cannot be set arbitrarily, but according to the technological constraints. For each water flow, minimum and maximum values are prescribed. Table 1 shows an example of the prescribed target temperatures and maximum water flows for continuous casting of the steel grade analyzed in this study. Minimum water flows are 0 for all zones and both center and corner positions.

Table 1: Target temperatures and water flow constraints for continuous casting of steel considered in numerical experiments

	Center positions		Corner positions	
Zone number	Target [°C]	Max flow [m^3/h]	Target [°C]	Max flow [m^3/h]
1	1050	50	880	50
2	1040	50	870	50
3	980	50	810	50
4	970	10	800	10
5	960	10	790	10
6	950	10	780	10
7	940	10	770	10
8	930	10	760	10
9	920	10	750	10

As we can see, coolant flow tuning in continuous casting of steel is a constrained two-objective optimization problem. To solve it by means of parallel multiobjective optimization, we integrated the optimization algorithm with a numerical simulator of the casting process. Given the coolant flow values, the simulator calculates the temperature field in the slab and extracts the values of objectives f_1 and f_2. For this purpose we use a numerical model of the process with Finite Element Method (FEM) discretization of the temperature field, and the corresponding nonlinear equations are solved with relaxation iterative methods. The model has previously been used in a single-objective optimization study of the casting process (Filipič and Laitinen, 2005), and in preliminary multiobjective optimization studies with the serial variant of the DEMO algorithm (Filipič et al., 2007).

Optimization calculations were performed for a selected steel grade with the slab cross-section of 1.70 m $\times$ 0.21 m. The assumed casting speed was 1.6 m/min, and the target core length, $l^{\text{core}*}$, 27 m.

4.3 Initial experiments and results

Initial parallel optimization experiments were performed to compare generational DEMO and AMS-DEMO. As shown in previous work (Filipič et al., 2007), solving the continuous casting optimization problem, DEMO appears to work best with population sizes between 20 and 40, which coincides well with the 32 processors available in the cluster. Therefore, a population size of 32 was chosen for the initial experiments. The remaining parameters for both parallel algorithms were set as follows: scaling factor F to 0.5, crossover probability to 0.1, DE scheme to rand/1/bin, and the stopping criterion to 9600 evaluations.

Figure 3: Example nondominated fronts after 9600 evaluations. Two fronts for separate runs are plotted to show the difference between typical final values of the hypervolume indicator I_H. Hypervolume indicator is explained in Subsection 4.4.

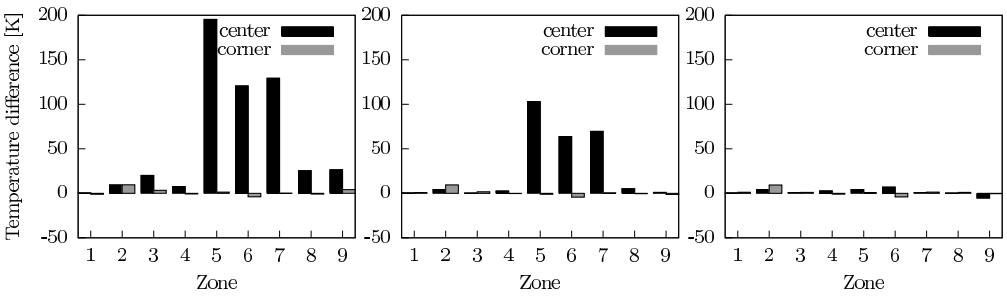

Figure 4: Resulting temperatures for three hand-picked solutions from the front with $I_H = 3.878$ in Figure 3, illustrating different tradeoffs between the two criteria. The leftmost chart presents the solution with the highest value of the first objective (512 K) and the lowest value of the second objective (0.0 m); the rightmost chart presents the solution with the lowest value of the first objective (41 K) and the highest of the second objective (2.7 m); and the chart in the middle presents the solution from the center of the front (first objective 257 K, second objective 1.3 m).

It turned out that both parallel algorithms were able to discover the solutions known from previous applications of the original DEMO (Filipič et al., 2007) demonstrating conformance with it. All nondominated fronts reached the size equal to the population size (set to 32). To illustrate the results, Figure 3 shows the resulting nondominated front (approximating the Pareto optimal front) found by generational DEMO. 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 – is evident from the presented nondominated front. 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. For example, the temperature difference for three solutions from the front displayed in Figure 3, two at the boundaries of the front and one from the middle of the front, are shown in Figure 4.

4.4 Convergence of AMS-DEMO

Convergence of AMS-DEMO was tested experimentally on 1, 2, 4, 8, 16, and 32 processors, where for each number of processors p, the algorithm was run 25 times. Generational DEMO was tested under the same conditions as AMS-DEMO, with the difference that each experiment was a batch of only 5 runs. Lower number of runs was used, in contrast to the 25 runs per AMS-DEMO experiment, because the number of processors p does not influence the way in which the algorithm works. Because the difference in the generational DEMO run times is always below one percent, 5 runs per each p were sufficient to calculate mean run times as a function of p. To compare the convergence rates of generational DEMO and the original DEMO, an additional batch of 25 runs of generational DEMO was performed on 32 processors. The presented comparisons of algorithms and of various numbers of processors were all performed using the mean

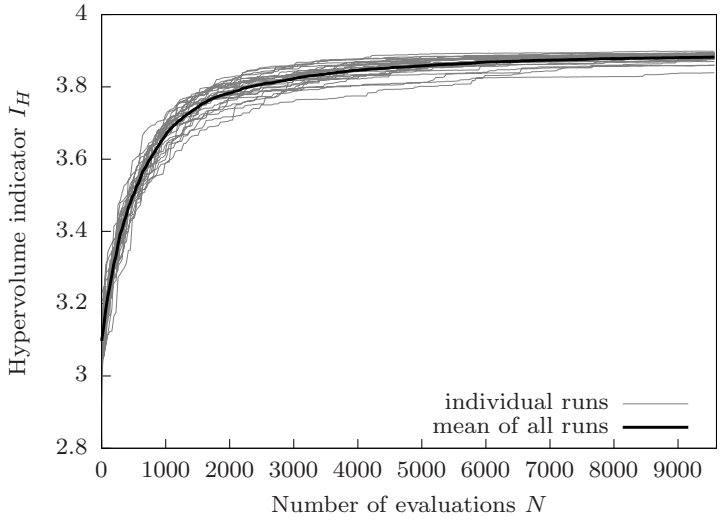

Figure 5: Algorithm convergence, indicated by the hypervolume indicator of individual runs, along with its mean value.

number of evaluations required to reach the same solution quality.

The convergence of the tested DEMO variants was evaluated using the hypervolume indicator I_H (Zitzler et al., 2002), also called the $\mathcal{S}$ metric (Zitzler and Thiele, 1998), which is a measure of the hypervolume of objective space that is dominated by a set of solutions. The properties of the hypervolume indicator (Knowles and Corne, 2002) enable observation of the convergence of solutions towards the optimum within a single run, and the comparisons of achieved solutions between two or more runs. On the other hand, the hypervolume indicator is sensitive to the properties of the nondominated front (Auger et al., 2009), such as the evenness of the distribution of solutions along the front, which makes comparison between different algorithms less reliable. The differences between the DEMO variants that we compare, however, are not in the variation operators, the truncation of solutions, or related functions of the algorithm, and therefore have no influence on the properties of the nondominated front, making comparison between the algorithms possible.

First, AMS-DEMO is evaluated against the original DEMO, with regards to the convergence of the algorithm, characterized by the convergence of the I_H. It should be noted that when the number of processors drops to one, AMS-DEMO reverts to original DEMO – given the same random generator and seed, the AMS-DEMO algorithm traverses the same path through the search space as the original DEMO, with no calculation overhead. Therefore, experiments with AMS-DEMO on a single processor are taken also as the experiments of the original DEMO. With the exception of the number of processors, no other algorithm parameter varied between the experiments. Population size was set to 32 and runs were terminated after 9600 (population size times 300) evaluations. Each run was carefully timed and the hypervolume indicator of the nondominated set of solutions was measured every 32 evaluations – after every population truncation. Values of the hypervolume indicator for the experiment on a single processor are shown in Figure 5.

Due to the changes in the algorithm required by the parallelization, the conver-

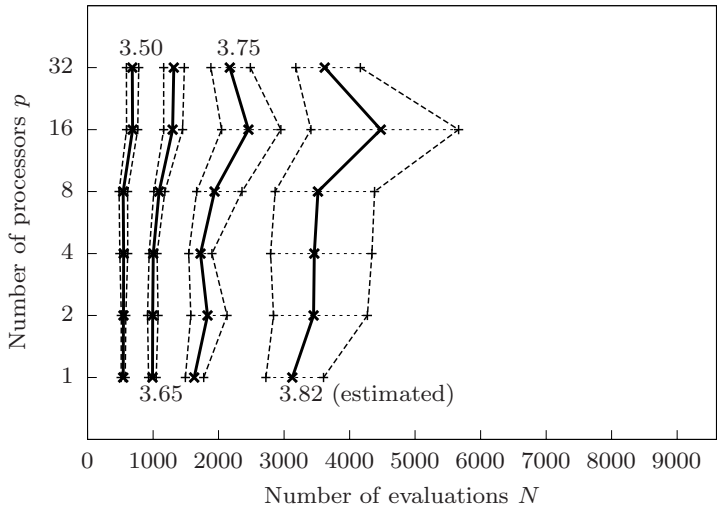

Figure 6: Mean numbers of evaluations required by AMS-DEMO for the population to reach a predefined I_H value. Best estimates for means are marked with x and connected with solid line, their 95% confidence intervals are marked with $+$, and connected with dashed lines. Predefined I_H values are shown with labels. Values for $I_H = 3.82$ are estimated from only 143 out of total 150 runs, because the other 7 runs did not reach this value in 9600 evaluations.

gence rate of AMS-DEMO is expected to slow down when the number of processors increases. The number of evaluations required to reach specific I_H values were examined and compared between the experiments. Mean values in Figure 6 indicate that increasing the number of processors does slow down the convergence. Their confidence intervals, which were calculated using the basic percentile bootstrap method (Efron, 1982), are quite large though, denoting low statistical confidence of such conclusions.

The statistical significance of the differences in number of evaluations performed by AMS-DEMO for relevant values of I_H is determined using the two-sample Kolmogorov–Smirnov test. As can be seen from Figure 7, the differences are statistically significant (P-value < 0.05) only for $p = \{16, 32\}$ and approximately for $I_H \in [3.26 \ldots 3.77]$. This is consistent with our expectations. The difference between DEMO and AMS-DEMO increases with increasing number of processors and is more important in the early stage of search when the convergence is faster, and less important in the later stage of search when the convergence is slower. The difference is also not detected immediately but rather after the initial random population is significantly improved.

The convergence rate of generational DEMO is also compared against the convergence rate of the original DEMO and the differences are found to be insignificant. Because generational DEMO runs equivalently on any number of processors p, we can conclude that it has an advantage over AMS-DEMO for larger numbers of p, where the convergence of AMS-DEMO is noticeably slower.

We finally analyze the selection lag. Figure 8 shows the distributions of selection lag on performed experiments. Means of distributions equal $pq-1$, as expected. Modes (peaks in distributions) also equal $pq - 1$ and distributions appear only slightly asymmetric. Although the total range of measured values is wide, e.g. from 6 to 58 for

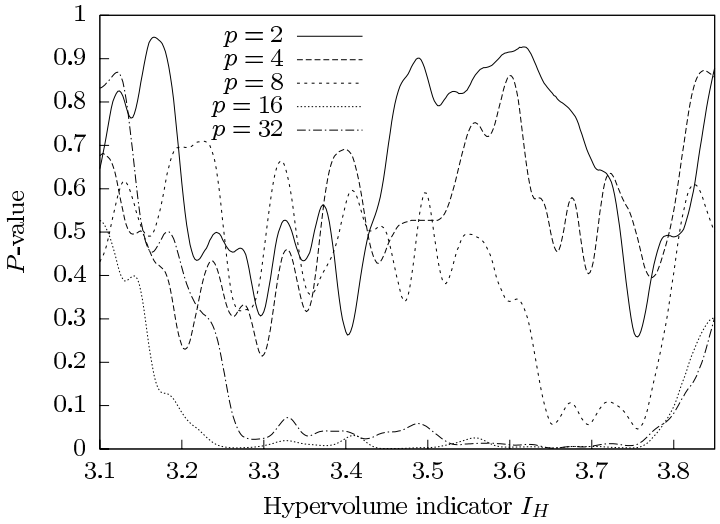

Figure 7: A significance test is performed for $I_H \in [3.1 \ldots 3.85]$. The null hypothesis is that using $p \in \{2, 4, 8, 16, 32\}$ processors does not require more evaluations than using only one processor. Graphs are filtered with Gauss filter with $\sigma = 0.02$ to reduce the noise in P-value and thus make individual curves more clearly visible.

$p = 32$, standard deviations are small, e.g. 2.2 for $p = 32$. Therefore, mean selection lag seems adequate to explain the difference in convergence between AMS-DEMO and DEMO.

Convergence tests can be summarized as follows. A bit surprisingly, generational DEMO converges at the same rate as the original DEMO, which should result in good parallel speedups. AMS-DEMO convergence slows down, as expected, as the number of processors increases. The change is only statistically significant at 16 and 32 processors however, and on less than the whole range of target solution qualities. Given much more than 25 runs per test, the change in AMS-DEMO convergence might be significant at other numbers of processors, but we are limited in the number of tests we make by the long execution times of tests, particularly those on smaller numbers of processors. Because selection lag is the driver of changes in convergence rate, we measure it on the performed tests. Its mean equals $pq - 1$ and its variability is low, both as expected. Because variability is low, the mean selection lag adequately explains the difference in convergence between AMS-DEMO and DEMO.

4.5 Speedup

In parallel computing, speedup S is a measure of how much faster a certain parallel algorithm is than the best sequential counterpart, or formally:

$$S(p) = \frac{t(1)}{t(p)} \,, \tag{6}$$

where $t(1)$ and $t(p)$ are the serial and parallel execution times, respectively. A measure related to speedup is efficiency, which is the speedup normalized with the number of processors:

$$E(p) = \frac{S(p)}{p} \,, \tag{7}$$

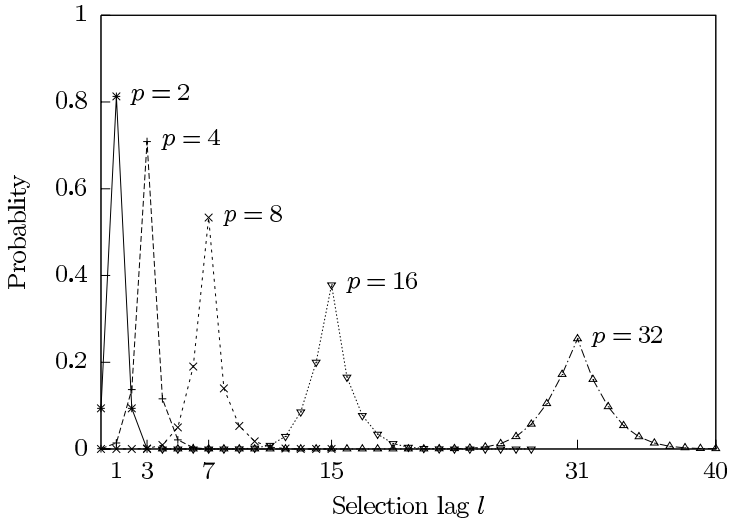

Figure 8: Probability distribution of the selection lag as a function of the number of processors p.

Efficiency is a measure of the level of utilization of computing resources for problem related tasks opposed to resources lost in parallelization overhead. Using speedup and efficiency, we address the relations between AMS-DEMO and the original DEMO, and between generational and the original DEMO, to get some deeper insight into the properties of the asynchronous versus synchronous parallelization.

When the tested algorithms are not equivalent, in the sense that they do not produce the same results, the execution times should be measured with care. The weak speedup presented in Subsection 4.1 is no longer appropriate to use, instead we use the speedup defined as type II.A speedup in Alba (2002), in which the execution times are the times spent by the compared algorithms to reach the same result quality, and we refer to it as *speedup*. We measure the result quality as I_H of the nondominated set of solutions and we therefore define the execution time of the algorithm running on p processors as $t(p, I_H)$. The speedup S of AMS-DEMO thus becomes the function of both p and I_H, and is formally writen as:

$$S(p, I_H) = \frac{t(1, I_H)}{t(p, I_H)} \,. \tag{8}$$

Figure 9 shows speedup on the range $I_H \in [3.10, 3.83]$.

To aid the analysis, the speedup was split to two factors:

$$S(p, I_H) = S_{\mathrm{c}}(p, I_H)\, S_{\mathrm{p}}(p) \,. \tag{9}$$

S_{p} is the speedup due to the increased computational resources provided by multiple processors, and S_{c} is the speedup due to the changes in the algorithm required by the parallelization. The former tells us how many times more computation is done per time unit on p processors in comparison to one processor, and the latter how many times less computational effort is required by the algorithm to reach solutions of similar quality on p processors than on one processor. It is evident from the experiments of algorithm convergence that increasing p increases the computational effort required by

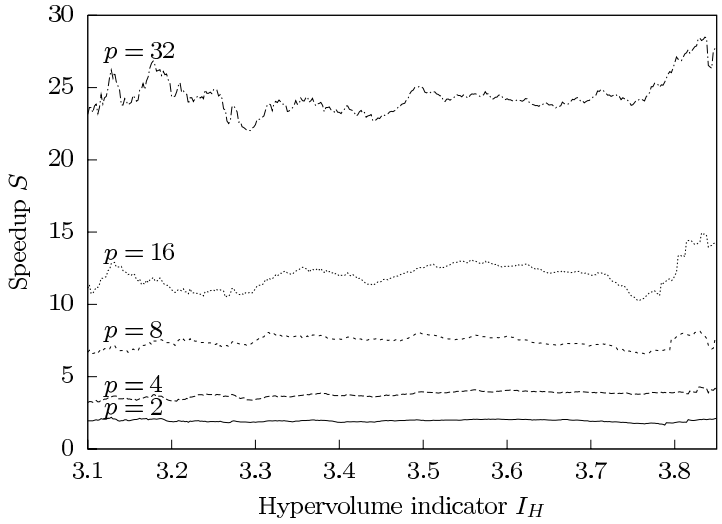

Figure 9: Speedup S relative to the target hypervolume indicator I_H value.

AMS-DEMO, so that S_c is expected to be less than 1. Computational effort is measured as the number of evaluations N_q required by the algorithm running on p processors to reach a predefined solution quality I_H; thus S_c can be calculated as:

$$S_\mathrm{c}(p, I_H) = \frac{N_q(p, I_H)}{N_q(1, I_H)} \,. \tag{10}$$

Finally, the speedup arising from the use of multiple processors, S_p, can be calculated as the ratio of the number of performed evaluations per time unit N_t on p processors to that on one processor:

$$S_\mathrm{p}(p) = \frac{N_t(p)}{N_t(1)} \,. \tag{11}$$

From Table 2 the average difference between AMS-DEMO and generational DEMO can be observed for $I_H \in [3.1 \ldots 3.85]$. The expected decrease in AMS-DEMO efficiency as a result of increasing number of processors is quantified in a column for S_c under AMS-DEMO. AMS-DEMO therefore becomes progressively less efficient than the original DEMO with every additional processor. On the other hand, high S_p – nearly equal to the number of processors on all the experiments – implies very good utilization of the available processors. The opposite holds for generational DEMO. While it is as efficient as the original DEMO at using evaluations, it is less efficient at utilizing additional processors, which is reflected in smaller S_p.

Looking at the speedups calculated from the performed experiments, an initial conclusion could be that the algorithms are closely matched, with generational DEMO having a slight advantage. The presented experiments, however, are biased, since they encompass only scenarios that particularly suit generational DEMO. Therefore we prepare two additional scenarios and analyze them both analytically and through additional experiments.

Table 2: Comparison of AMS-DEMO and generational DEMO in terms of speedups

	AMS-DEMO				Generational DEMO			
p	$S_{\mathrm{c}}(p)$	$S_{\mathrm{p}}(p)$	$S(p)$	$E(p)$	$S_{\mathrm{c}}(p)$	$S_{\mathrm{p}}(p)$	$S(p)$	$E(p)$
1	1	1	1	1	1	1	1	1
2	0.97	2.02	1.95	0.98	1	1.9	1.91	0.96
4	0.98	3.86	3.77	0.94	1	3.74	3.75	0.94
8	0.93	8.01	7.42	0.93	1	7.26	7.27	0.91
16	0.78	15.4	12	0.75	1	13.9	13.9	0.87
32	0.80	30.6	24.4	0.76	1	27.1	27.1	0.87

4.6 Analytical comparison

AMS-DEMO would exhibit a clear advantage over generational DEMO if the population size were not a multiple of the number of processors. Generational evolutionary algorithms, implementing the basic master-slave parallelization, evaluate one population at a time in parallel. But they cannot proceed to the next generation until the whole population is evaluated. Consequently, assuming that the evaluation of a single solution requires one processor, the number of processors used cannot exceed the size of the population. Furthermore, it is inefficient to use a number of processors that does not divide the size of the population. Consider the worst case scenario in which the population size n is larger than the number of processors by one: $n = p + 1$. Evaluating such a population would require $p - 1$ processors to each evaluate one solution and one processor to evaluate two solutions, resulting in considerable idle time of the processors. They would also be idle if equal numbers of solutions for evaluation were assigned to them, but the evaluation times of these solutions were not equal. Two possible (non-exclusive) causes of differences in evaluation times are a heterogeneous set of processors and an evaluation function with non-constant run time – dependent on the solution or simply random. The noted caveats of the master-slave parallelization apply to generational DEMO without exception. AMS-DEMO circumvents both of those caveats, with an important and rather surprising consequence – AMS-DEMO is able to utilize more processors than is the size of the population. We will explore how effective it is in this respect in Subsection 4.7.

To estimate the behavior of AMS-DEMO and (for reference) generational DEMO, equations for the execution time have been devised for both algorithms.

The generational DEMO execution time t_{gen} equals the number of generations N/n times single generation processing time. The single generation processing time is dominated by the evaluation of the population, which is a parallel evaluation of p solutions on p processors, repeated $\lfloor n/p \rfloor$ times, plus a parallel evaluation of the remaining $n \bmod p$ solutions on p processors:

$$t_{\mathrm{gen}} = \frac{N}{n} \left(t_{\mathrm{par}}(p) \left\lfloor \frac{n}{p} \right\rfloor + t_{\mathrm{par}}(n \bmod p) \right) . \tag{12}$$

The parallel evaluation time of m solutions is the expected value of the maximum of m evaluation times t_e:

$$t_{\mathrm{par}}(m) = E\left(\max_{i=1}^{m} \{t_{e,i}\} \right) . \tag{13}$$

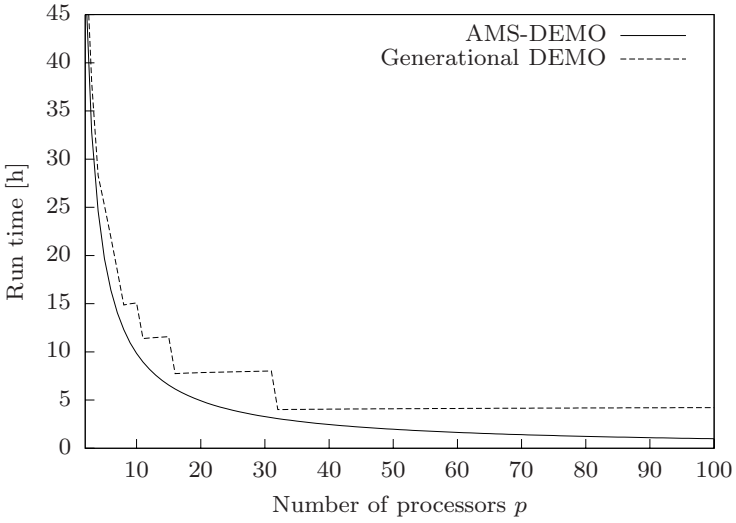

Figure 10: Estimated execution times for generational and AMS-DEMO, both set to terminate after 9600 evaluations.

It is approximated as the mean of the cumulative distribution function (CDF) of the maximum time of m evaluations, which equals the CDF of the solution evaluation time, raised to the power of m.

AMS-DEMO execution time is calculated as the sum of execution times of individual processors divided by the number of processors p. The sum of execution times of individual processors equals the sum of all evaluation times (approximately $\overline{t_e}\ N$) plus the sum of all idle times. Processors are only idle at the end of the optimization, from the time at which the first processor has finished its last evaluation, until the time when the last processor has finished its last evaluation. It can then be reasoned that one processor has 0 idle time (the one that is the last to finish) and the other $p-1$ processors experience average idle time of $0.5\ \overline{t_e}$.

$$t_{\mathrm{AMS}} = \frac{\overline{t_e}N + 0.5(p-1)\overline{t_e}}{p}\ . \tag{14}$$

The computed execution times were tested against the experimentally measured execution times and were found to be within the experimental confidence intervals. Using the given equations for the execution time of generational DEMO and AMS-DEMO and the measured evaluation times, the execution times were estimated for the number of processors on interval $[1, 100]$ and are shown in Figure 10.

From the execution times, speedups can be derived accurately for generational DEMO, and less accurately for AMS-DEMO, because the latter behaves differently for the different numbers of processors. Nevertheless, it can be argued that, for example, increasing the number of processors from 16 to 30, the execution time and the behavior of generational DEMO would not change, causing the speedup to drop significantly. AMS-DEMO, on the other hand, would experience two counteracting effects – the ability to run the same number of evaluations in less time, balanced out to some extent by the requirement for more evaluations to reach the same solution quality. As the experiments show, when increasing the number of processors the first effect always outweighs the second, yielding an increase in speedup. Therefore, increasing the num-

ber of processors is always beneficial for the performance of AMS-DEMO, while it often degrades the performance of generational DEMO.

4.7 Varying the queue size

Although queues have been implemented to reduce the slave idle time to a minimum, they also allow simulating more processors than are available on the experimental architecture. This allows for a simulation of an interesting algorithm property – the ability to run on a number of processors that is larger than the population size. Although there are other possibilities of simulating additional processors, e.g. running multiple processes on a single processor, using the queues is chosen because it simultaneously provides an example of the drawback of queues.

The algorithm running on p slaves, each having a queue of size q, explores the objective space in a similar fashion as if it were running on p times q slaves, each having a queue of size 1. This is because the algorithm behavior changes with the selection lag, for which was shown that its mean equals $pq - 1$. The same mean selection lag may be obtained through different values of p and q, therefore, increasing queue size emulates the use of additional processors. Although settings of the algorithm that produce the same mean selection lag produce a very similar behavior, running the algorithm on fewer processors with longer queues differs from running it on more processors with shorter queues. If a number of solutions are inserted into a single queue, they undergo selection in the order of insertion. On the other hand, if the same number of solutions are distributed between different processors and the evaluation time varies, they are likely to undergo selection in a different order. The additional out-of-order selection manifests as an increase in the selection lag variance, and, although difficult to quantify, has some influence on the algorithm behavior. In our experiments, for example on 32 processors, the mean selection lag is 31, while its standard deviation is 2.65, which we believe is small enough to be negligible.

Queue sizes of 10 and 20 are used on 32 processors, simulating 320 and 640 processors respectively, and the results are compared to the original DEMO. All the experiments are performed using the same algorithm parameters as before, including the population size of 32. The Speedup calculated from (8) and (14) is plotted in Figure 11.

First, we see from Figure 11 that the speedup grows much more slowly than the number of processors, but it grows nevertheless. Because the number of processors is not limited by the population size, the algorithm is able to produce speedups far greater than the population size. The drop in efficiency, however, should be taken into consideration when this property of the algorithm is used. The second observation is, that the speedup improves as the target I_H value rises. This follows from the property of AMS-DEMO that its traversal through the search space deviates the most from the original DEMO traversal when the convergence is fastest. Because the convergence of the algorithm slows at higher I_H, AMS-DEMO behaves more like the original DEMO and thus becomes more efficient, causing the speedup to increase.

4.8 Experiments on Grid'5000

To demonstrate the flexibility of AMS-DEMO, we performed additional experiments on the Grid'5000 (Bolze et al., 2006) computer setup. Grid'5000 is a research tool for studying large-scale distributed systems and high-performance scientific computing. It is distributed between nine sites, with each site hosting one or more clusters comprising several hundred processors. We used clusters *Bordereau* and *Bordeplage*, located at the *Bordeaux* site to perform four experiments with varying $p \in \{100, 200, 300, 400\}$, each

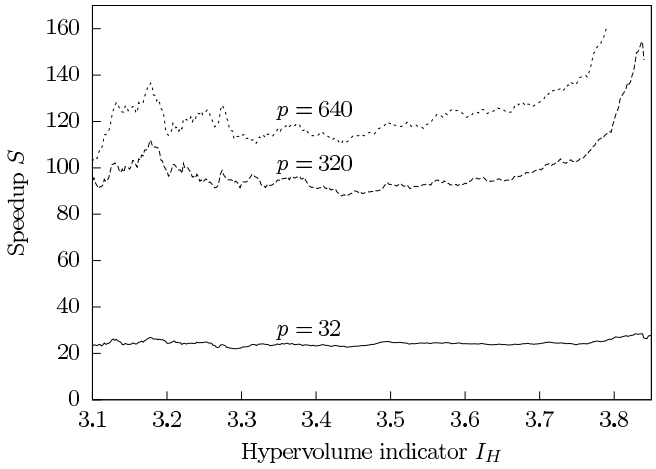

Figure 11: Simulated speedup relative to the target hypervolume indicator I_H value. Graphs are plotted up to the minimal hypervolume indicator value reached by at least 20 out of 25 runs, which lowers with higher p. Measured speedup on $p = 32$ is added for reference.

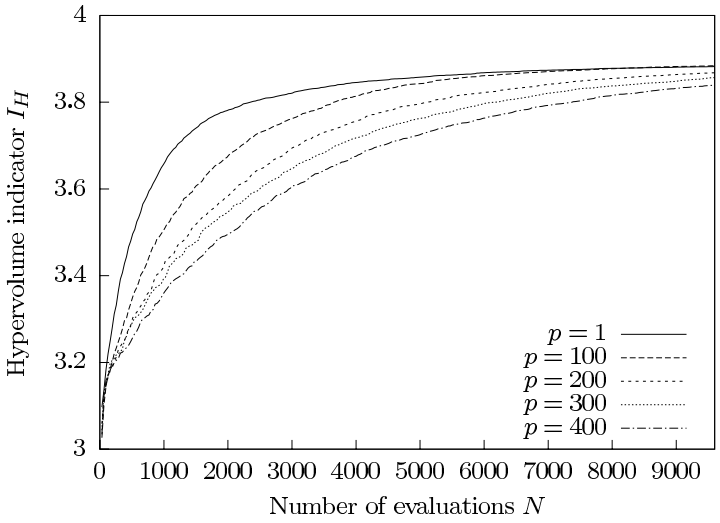

Figure 12: Hypervolume indicator I_H as a function of number of evaluations N, for varying number of processors p on Grid'5000.

repeated 25 times. Queue length was set to 1 and other algorithm settings were set as for the previous experiments.

The convergence of AMS-DEMO on Grid'5000 is compared to AMS-DEMO running on a single processor. The results are shown in Figure 12 . While the convergence rate gets slower with increasing p, the solutions reach about the same average I_H both on $p = 100$ and $p = 1$ within the performed number of evaluations. For $p > 100$, this does not happen, which confirms the results of the experiments with emulated processors.

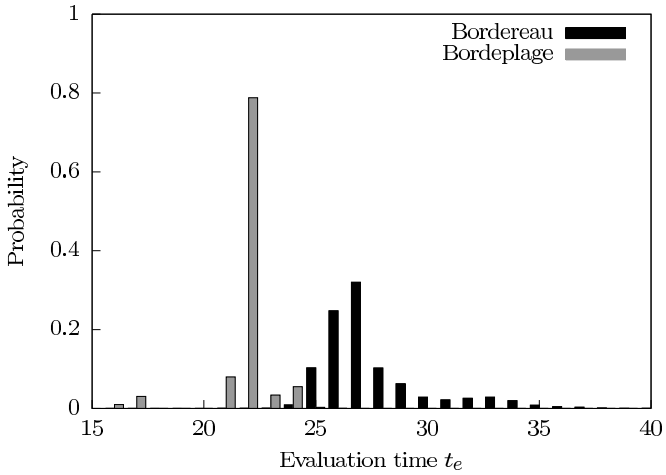

Figure 13: Probability distribution of the evaluation time t_e on the two clusters from Grid'5000 used in the experiments.

Clusters used in the experiments are comprised of various hardware and perform differently on the evaluation function. As seen from Figure 13, where the distribution of evaluation time t_e is plotted for all evaluations performed during experiments, Bordeplage performs evaluations faster than Bordereau. Its mean t_e is 21.8 s, compared to Bordereau's 27.9 s – it is faster by 22 % on average. If generational DEMO were considered, the difference in mean performance alone would cause poor utilization of the faster processors. Observing both clusters together, there is also the wide spread of t_e, between 16 s and 40 s, that would further lower processor utilization on generational DEMO. Another obstacle to the generational DEMO efficiency is a relatively small population size n, compared to the number of available processors. If n were fixed to 32, then generational DEMO would not be able to use additional processors of Grid'5000 at all, but if n was equal to p, its convergence would slow down, and more importantly, because of different population sizes, the results of generational DEMO could not be compared to the results of AMS-DEMO using hypervolume indicator. Therefore we do not experiment with generational DEMO on Grid'5000.

We calculate speedups according to (8), using the runs of AMS-DEMO running on one processor of the faster Bordeplage cluster and plot them in Figure 14. The results are similar to the results obtained by emulating processors, which were shown in Figure 11. Speedups are larger than n, and increase with I_H. We also observe that at $p = 400$, AMS-DEMO produces speedup larger than at $p = 300$ only for $I_H \gtrapprox 3.25$. This happens because the additional processors at $p = 400$ compared to $p = 300$ are mostly those from the slower cluster. Numbers of processors from each cluster, mean efficiencies $E(p)$ and mean speedups S, S_c, and S_p for each tested p are summarized in Table 3. Weighted speedup S_w is also given, which is the speedup of the algorithm performance on a set of p processors relative to the algorithm performance on an average processor from the same set, calculated as:

$$S_w(p) = \frac{t_{\text{Bordeplage}}(1)p_{\text{Bordeplage}} + t_{\text{Bordereau}}(1)p_{\text{Bordereau}}}{t(p)(p_{\text{Bordeplage}} + p_{\text{Bordereau}})} \,. \tag{15}$$

Weighted speedup is used in place of speedup to calculate efficiency $E(p)$ using (7). As

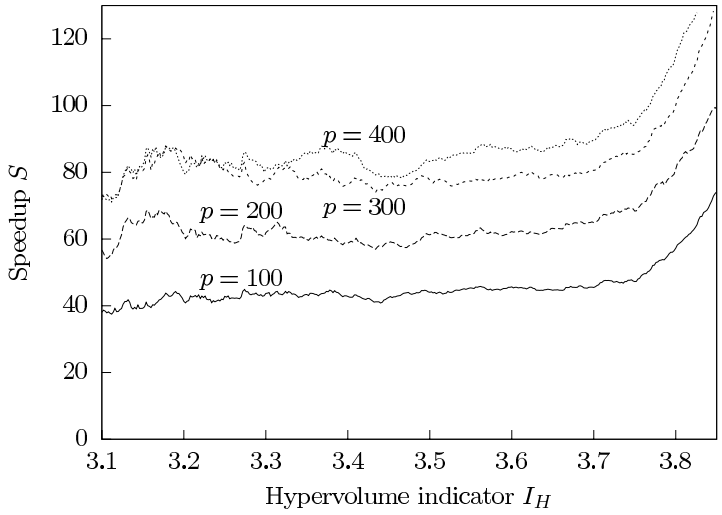

Figure 14: Speedup S as a function of hypervolume indicator I_H on Grid'5000.

Table 3: Comparison of speedups and efficiencies on Grid'5000

p							
$Total$	$Bordeplage$	$Bordereau$	$S_c(p)$	$S_p(p)$	$S(p)$	$S_w(p)$	$E(p)$
100	24	76	0.62	73.69	45.58	52.62	0.53
200	40	160	0.44	145.95	64.75	74.90	0.37
300	60	240	0.38	214.67	82.39	95.57	0.32
400	63	337	0.35	252.33	87.30	102.54	0.26

a result, $E(p)$ can be interpreted in the same way as in experiments on homogeneous parallel computer architecture, that is, as the ratio of the computational resources used in problem-related computation to the total computational resources used. Comparing Tables 2 and 3 we see that efficiency drops with larger numbers of processors. The main reasons are increased ratio of idle time to total execution time on the slaves because of the smaller number of evaluations each slave performs (also noticeable as a drop in S_p), and increased computational effort of the algorithm because of the larger number of slaves (indicated by a low S_p compared to p).

Using the data from all performed AMS-DEMO runs, we also analyzed the selection lag on a heterogeneous computer architecture of Grid'5000. We show the distribution of selection lag for tested values of p in Figure 15 and the relevant statistics in Table 4. Typical of the measured selection lag distributions is that they have a two-peak shape and are skewed to the right – distribution means are to the right of both peaks. The two peaks correspond to the two types of processors in the used portion of Grid'5000. The smaller peak at lower selection lags is produced by the lower number of Bordeplage processors, while the larger peak at higher selection lags is produced on the higher number of Bordereau processors. The most likely reason for skewness is the similarly skewed distribution of t_e on the Bordereau cluster (see Figure 13). Distributions are also very wide, as seen from their ranges and standard deviations compared

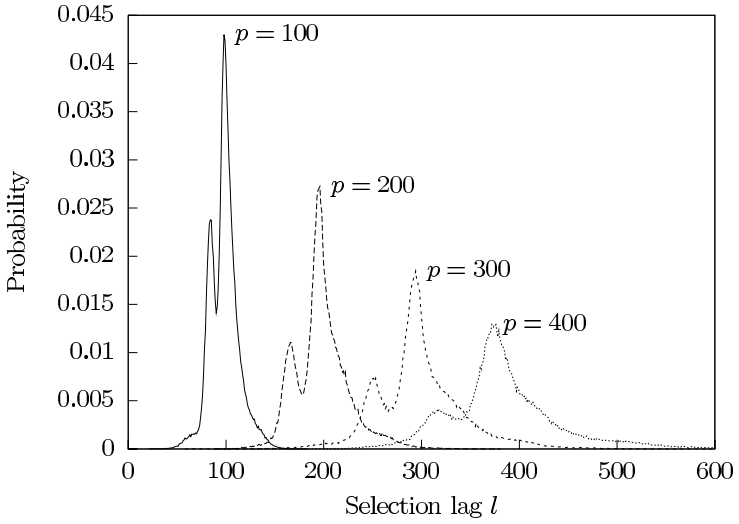

Figure 15: Selection lag l distribution for tested values of p on Grid'5000. Although selection lag values span between 23 and 1083, they are plotted between 0 and 600 for clarity.

Table 4: Selection lag statistics on Grid'5000

p	Range	Mean	Std. dev.	Peaks	
100	22 – 148	99	15	84	98
200	90 – 382	199	27	166	196
300	59 – 938	299	44	251	294
400	88 – 1082	399	91	316	372

to their means. Therefore, the mean selection lag may no longer provide enough information to fully understand the changes in AMS-DEMO convergence rate; further experiments are required to determine the effects that large variations in selection lag have on the AMS-DEMO convergence.

5 Conclusion

The steady-state Differential Evolution for Multiobjective Optimization (DEMO) algorithm was parallelized using an asynchronous master-slave parallelization type, creating the Asynchronous Master-Slave DEMO (AMS-DEMO). AMS-DEMO utilizes queues for each slave, which reduce the slave idle time to a negligible amount. Because of its asynchronous nature, the algorithm is able to fully utilize heterogeneous computer architectures and is not slowed down, even if the evaluation times are not constant.

Unlike the more common synchronous master-slave parallelization of generational algorithms, which traverse the decision space identically on any number of processors, the asynchronous master-slave parallelization changes the trajectory in which the algorithm traverses the decision space. Selection lag – a property that fully characterizes this change – was identified. Selection lag depends directly on the number of proces-

sors and queue sizes, and has an adverse effect on the algorithm, increasing the number of evaluations required to find optimal solutions. Experiments on a real-world problem indicate that the effect of selection lag is negligible for a number of processors lower than about half the population size. Although AMS-DEMO convergence rate appears to deteriorate slightly on such numbers of processors, we did not find this deterioration statistically significant. Only for larger numbers of processors does the increase in the number of evaluations become statistically significant. Nevertheless, we find that, when increasing the number of processors, the requirement for additional evaluations caused by the increased selection lag is outweighed by the additional computational resources provided by the processors, resulting in shorter optimization times and larger speedups. This finding is robust, and holds for all the performed experiments, even with numbers of processors up to several times the population size.

The constraints for the number of processors were also reduced, compared to the constraints imposed by the synchronous master-slave parallelization. The number of processors is not required to divide the population size and may even exceed it. Experiments on the computers of Grid'5000 showed that on such numbers of processors, AMS-DEMO achieves speedups larger than population size and therefore larger than theoretical limit for generational algorithms. Although we used no grid-computing middleware for our tests on Grid'5000, extending AMS-DEMO to use it should be possible. Asynchronous nature makes AMS-DEMO robust to communication failures and able to handle dynamic allocation of processing resources, and thus suitable for grid computing. However, additional work is needed to explore the behavior of AMS-DEMO in the presence of failures and on the grid.

We tested the AMS-DEMO algorithm on a benchmark problem and a real-life problem. On the benchmark problem SYMPART, the evaluation function is simple to compute, and we inserted a variable delay, to observe how the weak speedup changes with evaluation time. As expected, AMS-DEMO was slower than the original DEMO in tests with extremely short evaluation time. If the evaluation time was comparable to communication time, AMS-DEMO performance improved while in tests with evaluation time several orders of magnitude longer than communication time, AMS-DEMO performed at near-linear weak speedup. We also devised a simplified model of weak speedup for approximate performance of AMS-DEMO.

The majority of the experiments was done on the real-life multiobjective optimization problem of continuous steel casting, which requires the optimization of parameters of the industrial procedure according to two objectives. As a result of a computationally demanding and time consuming evaluation function, which is based on a computer simulation, this problem is difficult to solve. Therefore, the parallelization of the optimization algorithm was used to make solving more manageable. The efficiency of the proposed AMS-DEMO algorithm was contrasted with the simpler and more straightforward synchronous master-slave parallelization method. The experiments reveal that the synchronous master-slave parallelism can be equally fast or slightly faster on a homogeneous architecture, even when the evaluation times are not constant. When conditions unfavorable to synchronous parallelism accumulate, however, AMS-DEMO gains advantage, as the experiments on a heterogeneous Grid'5000 architecture show.

Although the predictions based on the analysis and the experimental results so far agree, AMS-DEMO should be further tested on other problems before making firm conclusions. Since the parallel properties of AMS-DEMO depend largely on the proposed asynchronous master-slave parallelization method and less so on the original DEMO algorithm, a sensible next step would be to investigate the proposed parallelization

type independently. Its applicability to other algorithms, both single- and multiobjective, would be of special interest. Finally a more in-depth understanding of the selection lag and new ways to minimize its negative effects remain topics for further work. The ways in which the selection lag distribution influences the algorithm convergence would be of most interest.

References

Akl, S. G. (1997). *Parallel Computation: Models and Methods*. Prentice Hall, Upper Saddle River.

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

Alba, E. and Troya, J. M. (2002). Improving flexibility and efficiency by adding parallelism to genetic algorithms. *Statistics and Computing*, 12(2):91–114.

Auger, A., Bader, J., Brockhoff, D., and Zitzler, E. (2009). Theory of the hypervolume indicator: Optimal μ-distributions and the choice of the reference point. In *Proceedings of the 10th Foundations of Genetic Algorithms Workshop – FOGA 2009*, pages 87–102, New York, NY, USA. ACM.

Bolze, R., Cappello, F., Caron, E., Daydé, M., Desprez, F., Jeannot, E., Jégou, Y., Lantéri, S., Leduc, J., Melab, N., Mornet, G., Namyst, R., Primet, P., Quetier, B., Richard, O., Talbi, E.-G., and Touche, I. (2006). Grid'5000: a large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494.

Cantú-Paz, E. (1997). A survey of parallel genetic algorithms. Technical report, University of Illinois at Urbana-Champaign.

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

Deb, K., Thiele, L., Laumanns, M., and Zitzler, E. (2005). Scalable test problems for evolutionary multi-objective optimization. In Abraham, A., Jain, R., and Goldberg, R., editors, *Evolutionary Multiobjective Optimization: Theoretical Advances and Applications*, pages 105–145. Springer-Verlag, New York.

Eberhard, P., Dignath, F., and Kübler, L. (2003). Parallel evolutionary optimization of multibody systems with application to railway dynamics. *Multibody System Dynamics*, 9(2):143–164.

Efron, B. (1982). *The Jackknife, the bootstrap and other resampling plans*. Regional Conference Series in applied mathematics. Society for Industrial and applied mathematics, Philadelphia.

Eiben, A. E. and Smith, J. E. (2003). *Introduction to Evolutionary Computing*. Springer-Verlag, Berlin, Heidelberg.

Filipič, B. and Depolli, M. (2009). Parallel evolutionary computation framework for single- and multiobjective optimization. In Trobec, R., Vajteršic, M., and Zinterhof, P., editors, *Parallel Computing – Numerics, Applications, and Trends*, pages 217–240. Springer, Dordrecht.

Filipič, B. and Laitinen, E. (2005). Model-based tuning of process parameters for steady-state steel casting. *Informatica*, 29(4):491–496.

Filipič, B., Tušar, T., and Laitinen, E. (2007). Preliminary numerical experiments in multiobjective optimization of a metallurgical production process. *Informatica*, 31(2):233–240.

Gropp, W., Lusk, E., Doss, N., and Skjellum, A. (1996). High-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828.

Huband, S., Barone, L., While, R. L., and Hingston, P. (2005). A scalable multi-objective test problem toolkit. In Coello, C. A. C., Aguirre, A. H., and Zitzler, E., editors, *Proceedings of the Third International Conference on Evolutionary Multi-Criterion Optimization – EMO 2005*, pages 280–295.

Kennedy, J. and Eberhart, R. (1995). Particle swarm optimization. In *Proceedings of IEEE International Conference on Neural Networks*, pages 1942–1948.

Knowles, J. and Corne, D. (2002). On metrics for comparing non-dominated sets. In *Proceedings of the 2002 Congress on Evolutionary Computation Conference – CEC'02*, pages 711–716. IEEE Press.

Koh, B.-I., George, A. D., Haftka, R. T., and Fregly, B. J. (2006). Parallel asynchronous particle swarm optimization. *International Journal for Numerical Methods in Engineering*, 67(4):578–595.

Lewis, A., Mostaghim, S., and Scriven, I. (2009). Asynchronous multi-objective optimisation in unreliable distributed environments. In Lewis, A., Mostaghim, S., and Randall, M., editors, *Biologically-inspired Optimisation Methods: Parallel Algorithms, Systems and Applications*, pages 51–78. Springer, Berlin.

Lim, D., Soon Ong, Y., Jin, Y., Sendhoff, B., and Sung Lee, B. (2007). Efficient hierarchical parallel genetic algorithms using grid computing. *Future Generation Computer Systems*, 23(4):658–670.

Luna, F., Nebro, A. J., and Alba, E. (2006). *Parallel Evolutionary Computations*, chapter Parallel Evolutionary Multiobjective Optimization, pages 33–56. Studies in Computational Intelligence, Parallel Evolutionary Computations. Springer-Verlag, Berlin Heidelberg.

Mostaghim, S., Branke, J., Lewis, A., and Schmeck, H. (2008). Parallel multi-objective optimization using master-slave model on heterogeneous resources. In *Proceedings of the 2008 Congress on Evolutionary Computation – CEC'08*, pages 1981–1987.

Nebro, A. J. and Durillo, J. J. (2010). A study of the parallelization of the multi-objective metaheuristic MOEA/D. In *Proceedings of the 4th international conference on Learning and intelligent optimization*, LION'10, pages 303–317, Berlin, Heidelberg. Springer-Verlag.

Nebro, A. J., Luna, F., Talbi, E.-G., and Alba, E. (2005). Parallel multiobjective optimization. In Alba, E., editor, *Parallel Metaheuristics*, pages 371–394. John Wiley & Sons, New Jersey.

Oliveira, L. S., R.Sabourin, Bortolozzi, F., and Suen, C. (2003). A methodology for feature selection using multi-objective genetic algorithms for handwritten digit string recognition. *International Journal of Pattern Recognition and Artificial Intelligence*, 17:2003.

Price, K., Storn, R. M., and Lampinen, J. A. (2005). *Differential Evolution: A Practical Approach to Global Optimization*. Natural Computing Series. Springer-Verlag, Berlin, Heidelberg.

Price, K. V. and Storn, R. (1997). Differential evolution: A simple evolution strategy for fast optimization. *Dr. Dobb's Journal*, 22(4):18–24.

Quagliarella, D. and Vicini, A. (1998). Sub-population policies for a parallel multiobjective genetic algorithm with applications to wing design. In *Proceedings of the 1998 IEEE International Conference On Systems, Man, and Cybernetics – SMC 1998*, pages 3142–3147, San Diego, California.

Radtke, P. V. W., Oliveira, L. S., Sabourin, R., and Wong, T. (2003). Intelligent zoning design using multi-objective evolutionary algorithms. In *Proceedings of the 7th International Conference on Document Analysis and Recognition – ICDAR 2003*, pages 824–828.

Robič, T. and Filipič, B. (2005). DEMO: Differential evolution for multiobjective optimization. In *Proceedings of the Third Conference on Evolutionary Multi-Criterion Optimization – EMO 2005*, volume 3410 of *Lecture Notes in Computer Science*, pages 520–533.

Sasaki, D., Morikawa, M., Obayashi, S., and Nakahashi, K. (2001). Aerodynamic shape optimization of supersonic wings by adaptive range multiobjective genetic algorithms. In *Proceedings of the First International Conference on Evolutionary Multi-Criterion Optimization – EMO'01*, pages 639–652.

Scriven, I., Irel, D., Lewis, A., Mostaghim, S., and Branke, J. (2008). Asynchronous multiple objective particle swarm optimisation in unreliable distributed environments. In *Proceedings of the 2008 Congress on Evolutionary Computation – CEC'08*, pages 2481–2486.

Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J. (1996). *MPI – The Complete Reference*. The MIT Press, Cambridge.

Stanley, T. J. and Mudge, T. (1995). A parallel genetic algorithm for multiobjective microprocessor design. In *Proceedings of the Sixth International Conference on Genetic Algorithms – ICGA 1995*, pages 597–604. Morgan Kaufmann Publishers.

Storn, R. and Price, K. V. (1997). Differential evolution: A simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11(4):341–359.

Suganthan, P. N. (2007). Performance assessment on multi-objective optimization algorithms. http://www3.ntu.edu.sg/home/epnsugan/.

Talbi, E.-G. and Meunier, H. (2006). Hierarchical parallel approach for GSM mobile network design. *Journal of Parallel and Distributed Computing*, 66(2):274–290.

Talbi, E.-G., Mostaghim, S., Okabe, T., Ishibuchi, H., Rudolph, G., and Coello Coello, C. A. (2008). Parallel approaches for multiobjective optimization. In Branke, J., Deb, K., Miettinen, K., and Slowinski, R., editors, *Multiobjective Optimization*, pages 349–372. Springer-Verlag, Berlin, Heidelberg.

Tušar, T. and Filipič, B. (2007). Differential evolution versus genetic algorithms in multiobjective optimization. In Obayashi, S., Deb, K., Poloni, C., Hiroyasu, T., and Murata, T., editors, *Proceedings of the Fourth International Conference on Evolutionary Multi-Criterion Optimization – EMO 2007*, Lecture Notes in Computer Science, pages 257–271, Matsushima, Japan. Springer, Berlin. LNCS, Vol. 4403.

van Veldhuizen, D. A., Zydallis, J. B., and Lamont, G. B. (2003). Considerations in engineering parallel multiobjective evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 7(2):144–173.

Zamuda, A., Brest, J., Bošković, B., and Žumer, V. (2007). Differential Evolution for Multiobjective Optimization with Self Adaptation. In *Proceedings of the 2007 Congress on Evolutionary Computation – CEC'07*, pages 3617–3624.

Zitzler, E., Deb, K., and Thiele, L. (2000). Comparison of multiobjective evolutionary algorithms: Empirical results. *Evolutionary Computation*, 8(2):173–195.

Zitzler, E. and Künzli, S. (2004). Indicator-based selection in multiobjective search. In *Proceedings of the Eight Conference on Parallel Problem Solving from Nature – PPSN VIII*, pages 832–842.

Zitzler, E., Laumanns, M., and Thiele, L. (2001). SPEA2: Improving the strength pareto evolutionary algorithm. Technical Report 103, Swiss Federal Institute of Technology (ETH) Zurich, Gloriastrasse 35, CH-8092 Zurich, Switzerland.

Zitzler, E. and Thiele, L. (1998). Multiobjective optimization using evolutionary algorithms – a comparative case study. In *Proceedings of the Fifth Conference on Parallel Problem Solving from Nature – PPSN V*, pages 292–301, Berling Heidelberg. Springer-Verlag.

Zitzler, E., Thiele, L., Laumanns, M., Fonseca, C. M., and da Fonseca, V. G. (2002). Performance assessment of multiobjective optimizers: An analysis and review. *IEEE Transactions on Evolutionary Computation*, 7(2):117–132.