PARALLELIZATION OF AN EVOLUTIONARY ALGORITHM FOR MULTIOBJECTIVE OPTIMIZATION

Matjaž Depolli

Doctoral Dissertation
Jožef Stefan International Postgraduate School
Ljubljana, Slovenia, August 2010

Evaluation Board:

Prof. Dr. Marko Bohanec, Chairman, Jožef Stefan Institute, Ljubljana, Slovenia
Assist. Prof. Dr. Jurij Šilc, Member, Jožef Stefan Institute, Ljubljana, Slovenia
Prof. Dr. Marjan Mernik, Member, Faculty of Electrical Engineering and Computer Science, University of Maribor, Slovenia

MEDNARODNA PODIPLOMSKA ŠOLA JOŽEFA STEFANA
JOŽEF STEFAN INTERNATIONAL POSTGRADUATE SCHOOL
Ljubljana, Slovenia

Matjaž Depolli

PARALLELIZATION OF AN EVOLUTIONARY ALGORITHM FOR MULTIOBJECTIVE OPTIMIZATION

Doctoral Dissertation

PARALELIZACIJA EVOLUCIJSKEGA ALGORITMA ZA VEČKRITERIJSKO OPTIMIZACIJO

Doktorska disertacija

Supervisor: Assoc. Prof. Dr. Bogdan Filipič
Co-supervisor: Assoc. Prof. Dr. Roman Trobec

July 2010

Contents

Abstract

Solving real-life optimization problems numerically is often very time demanding, because of high complexity of the simulations that are usually involved. Solving such problems becomes highly impractical for this reason and can even lead to use of less complex and also less accurate models. Fortunately, evolutionary algorithms, often used in numerical optimization, can be parallelized with relative ease, which significantly reduces the time required for optimization on parallel computer architectures.

Parallelizing DEMO – an evolutionary algorithm for multiobjective optimization, AMS-DEMO is created. The algorithm was designed for solving time demanding problems efficiently on both homogeneous and heterogeneous parallel computer architectures. To maximize flexibility and robustness, it uses a master-slave parallelization method that is modified to allow for asynchronous communication between computers.

AMS-DEMO performance is analyzed on two complex real-life multiobjective optimization problems and compared against a simpler but related algorithm, parallelized using the conventional master-slave method. Experimental tests are performed on two different parallel setups, one homogeneous and one heterogeneous, while the theoretical analysis extends the test results to cover a few hypothetical setups as well. AMS-DEMO is found to be very flexible and can be efficiently used on heterogeneous computer architectures. On homogeneous architectures and problems with constant-time fitness evaluation functions, however, the same performance as with AMS-DEMO can be achieved with much simpler parallel algorithms.

Selection lag is identified as the key property of evolutionary algorithms parallelized using asynchronous master-slave method. It explains how the behavior of the algorithm changes depending on parallel computer architecture, mainly on the number of processors that a given architecture offers. The dependence of algorithm efficiency on selection lag is shown, completing the link between efficiency and the number of processors in the parallel computer running the algorithm.

Povzetek

Med pogostejše oblike problemov, ki jih rešujemo z računalniki, sodijo optimizacijski problemi. To so problemi, ki zahtevajo iskanje najboljše rešitve izmed množice možnih rešitev po podanem kriteriju. Če je množica rešitev neskončna ali pa tako velika, da je ni mogoče preiskati v razumnem času, potem reševanje problema zahteva uporabo stohastičnih metod, ki ne najdejo vedno absolutno najboljših rešitev, najdejo pa dobre rešitve, katerih kakovost je tem boljša, čim več časa namenimo iskanju. Zato je pohitritev optimizacije z uporabo vzporednih računalniških arhitektur zelo dobrodošla. V disertaciji obravnavamo nov algoritem za optimizacijo, ki omogoča učinkovito uporabo vzporednih računalniških arhitektur, tudi takih z različnimi računalniki, in dosega zelo visoke pohitritve.

Uvod

V praksi pogosto naletimo na optimizacijske probleme, ki zahtevajo optimizacijo po več kriterijih hkrati. Kriteriji si največkrat tudi nasprotujejo, kar pomeni, da izboljšanje rešitve po enem kriteriju povzroči njeno poslabšanje po vsaj enem od ostalih kriterijev. Taki problemi zato nimajo le ene optimalne rešitve, temveč množico rešitev, ki ji rečemo Pareto optimalna množica, njeni predstavitvi v prostoru kriterijev pa Pareto optimalna fronta. Za vsako rešitev iz Pareto optimalne množice velja, da ne obstaja nobena druga rešitev, ki bi bila po vseh kriterijih vsaj enako dobra ali boljša.

Tovrstne probleme učinkovito rešujemo z evolucijskimi algoritmi (EA) za večkriterijsko optimizacijo, angl. multi-objective evolutionary algorithms (MOEA). To so populacijski algoritmi (delujejo nad množicami možnih rešitev), ki optimirajo z mehanizmi privzetimi iz biološke evolucije (selekcija, mutacija, reprodukcija z rekombinacijo) [74, 14, 71, 72, 1, 19, 20, 2, 55, 21].

Mnogi praktični problemi postanejo zaradi velike računske zahtevnosti skoraj nerešljivi na osebnih računalnikih. Obstajajo zmogljivejši vzporedni računalniki, katerih skupna lastnost je uporaba več procesorjev, a za ustrezen izkoristek njihovih zmogljivosti potrebujemo posebej prirejene vzporedne algoritme. Vzporedni računalniški sistemi se glede načina komunikacije delijo na sisteme s skupnim pomnilnikom in na take, ki si pomnilnika ne delijo in zato procesorji komunicirajo s sporočili preko računalniškega omrežja. Najbolj razširjeni so slednji, kjer sta dve najpogostejši podvrsti omrežje (angl. grid) in gruča (angl.

cluster). Za omrežje je značilna heterogenost strojne in programske opreme na vozliščih (osnovnih gradnikih sistema, največkrat so to samostojni računalniki) ter komunikacije med vozlišči prek povezav, ki si jih delijo z drugimi uporabniki, na primer interneta in lokalne mreže. Skupki so navadno homogeni, povezani s hitrimi namenskimi povezavami, kot so vodila ali mreže, ki omogočajo vozliščem hkratno povezavo z več sosedi.

Neodvisno od tipa arhitekture vzporednih sistemov, vzporedni algoritmi lahko tečejo na več procesorjih hkrati, zato morajo biti sestavljeni iz med seboj razmeroma neodvisnih delov (sklopov). To so sklopi, ki za svojo izvedbo potrebujejo čim manj podatkov iz drugih sklopov in zato malo komunikacije. Obstajajo standardni postopki paralelizacije, katerih izbira je odvisna od algoritma, ki ga paraleliziramo, in ciljne računalniške arhitekture [65, 62, 33, 8, 36, 40, 44].

Paralelizacija evolucijskih algoritmov za večkriterijsko optimizacijo

Nekateri avtorji na področju evolucijskih algoritmov za večkriterijsko optimizacijo opozarjajo na pomanjkanje razvoja vzporednih algoritmov [16, 67, 15], kljub temu, da evolucijski algoritmi spadajo med algoritme, ki jih lahko zelo dobro paraleliziramo in s tem pohitrimo iskanje rešitev [12, 6, 5, 41, 13]. MOEA od EA sicer podedujejo splošne načine paralelizacije, a se razlikujejo v podrobnostih [50, 69, 22, 67, 18, 60]. Trije splošni (osnovni) načini paralelizacije evolucijskih algoritmov so:

- Način *nadrejeni-podrejeni* (angl. master-slave). Procesorje, na katerih teče algoritem, razdelimo na podrejene in enega nadrejenega. Nadrejeni izvaja celoten algoritem, medtem ko podrejeni sočasno vrednotijo rešitve. V osnovi lahko algoritem paraleliziran po načinu nadrejeni-podrejeni preiskuje prostor rešitev enako kot zaporedni algoritem, a pogosto za ceno slabe izrabe procesorjev.

- *Otoški model* (ang. island model). Na vsakem sodelujočem procesorju (otoku) teče algoritem podoben zaporednemu, ponavadi na manjši populaciji. Zato je ta način paralelizacije večpopulacijski. Otoki komunicirajo med seboj v časovnih intervalih in si s tem izmenjujejo do tedaj najdene najboljše rešitve. Izraba procesorjev je dobra, komunikacije je malo, vendar tudi pohitritev ni nujno velika. Odvisna je od problema, ki ga rešujemo, ter v precejšnji meri od števila procesorjev.

- *difuzijski model* (angl. diffusion model). Sodelujoči procesorji si razdelijo populacijo na več majhnih podpopulacij, med katerimi poteka kombiniranje rešitev le znotraj sosednjih podpopulacij. Tako kot otoški je tudi difuzijski model večpopulacijski. Procesorji izvajajo algoritem podoben zaporednemu, a na svoji populaciji, ki jo kombinirajo z rešitvami iz populacij sosednjih procesorjev. Izraba procesorjev je dobra, komunikacije je veliko, a je lokalna (med vnaprej določenimi procesorskimi

pari), zato je učinkovita uporaba modela omejena na nekaj topologij povezave procesorjev, značilnih le za velike in dobro povezane računalniške gruče, ki lahko komunikacijo razporedijo med svoje številne povezave med procesorji in tako zmanjšajo delež časa potrebnega za komunikacijo.

Osnovni paralelizacijski načini ponujajo različne kompromise med stopnjo preiskanosti prostora rešitev, izrabo procesorskih zmogljivosti in pohitritvijo. Možna je tudi uporaba hibridnega načina, kjer z kombinacijo dveh ali več osnovnih načinov bolje izkoristimo njihove prednosti in zmanjšamo vpliv njihovih slabih strani.

Pristop nadrejeni-podrejeni ponuja identično preiskovanje prostora rešitev kot zaporedni algoritem, a pri generacijskih algoritmih ne more popolnoma izkoristiti vseh procesorjev, ki so na voljo, ker procesorji izgubljajo čas s čakanjem drug na drugega, s čimer se zagotavlja časovno usklajeno delovanje. Z zamenjavo generacijskega algoritma z algoritmom s stabilnim stanjem (angl. steady-state) in z dopolnitvami algoritma, katere opisujemo v disertaciji, je mogoče izkoriščenost procesorjev zelo izboljšati. Predmet disertacije sta razvoj takega algoritma in njegovo vrednotenje na zahtevnih realnih problemih.

Algoritem AMS-DEMO

Na podlagi Diferencialne evolucije za večkriterijsko optimizacijo, angl. Differential Evolution for Multiobjective Optimization (DEMO), smo razvili vzporedni algorithm AMS-DEMO (Asynchronous Master-Slave DEMO). AMS-DEMO po pristopu nadrejeni-podrejeni razdeli delo med enega nadrejenega in več podrejenih procesov. Nadrejeni proces opravlja vse delo, kot izvirni DEMO, razen vrednotenja rešitev. Vsako na novo ustvarjeno rešitev, bodisi ustvarjeno naključno, v začetni generaciji, bodisi ustvarjeno s pomočjo variacijskih operatorjev, v naslednjih generacijah, nadrejeni proces pošlje enemu izmed podrejenih v vrednotenje. Podrejeni procesi le sprejemajo rešitve od nadrejenega, jih vrednotijo in rezultate vračajo nadrejenemu. Pri tvorjenju novih rešitev in pri sprejemanju rezultatov vrednotenja se nadrejeni obnaša kot izvirni DEMO, ki je algoritem s stabilnim stanjem. To pomeni, da nove rešitve tvori iz aktivne populacije in ovrednotene rešitve vključuje v aktivno populacijo – ne dela torej menjav celotnih geeracij, kot to počno generacijski algoritmi. AMS-DEMO se od izvirnega algoritma razlikuje v tem, da se lahko med vrednotenjem ene rešitve aktivna populacija spremeni; v populacijo se namreč v tem času lahko vključi rešitev, ki je bila ovrednotena na drugem procesorju. Ta sprememba vpliva na konvergenco algoritma AMS-DEMO v primerjavi z izvirnim algoritmom DEMO.

Nadrejeni proces hrani podatke o številu poslanih rešitev in prejetih rezultatov vrednotenj za vsakega od podrejenih procesov, s katerimi si pomaga pri odločanju kdaj naj tvori nove rešitve in komu naj jih pošlje. Njegov cilj je, da so vsi podrejeni obremenjeni ves čas delovanja, kar pomeni, da vsak podrejeni po končanem vrednotenju in odposlani rešitvi čaka čim manj časa na novo rešitev za vrednotenje. K uresničitvi tega cilja pri-

pomorejo čakalne vrste v podrejenih procesih, ki hranijo rešitve prejete od nadrejenega, in s tem omogočijo, da podrejeni takoj po končanem vrednotenju lahko začne z naslednjim. Komunikacija med nadrejenim in podrejenim procesom zato postane popolnoma asinhrona, kar pomeni, da tako prvi kot drugi lahko pošljeta sporočilo in takoj nadaljujeta z delom, brez čakanja na odgovor. Ker je nadrejeni proces obremenjen razmeroma malo v primerjavi s podrejenimi procesi, ne potrebuje procesorja le zase, ampak se izvaja na istem procesorju kot eden izmed podrejenih procesov. Ostali podrejeni procesi tečejo vsak na svojem procesorju.

Lastnost algoritma AMS-DEMO, ki določa stopnjo odstopanja od izvirnega algoritma, poimenujemo *zakasnitev v selekciji*. Za algoritem je definirana kot porazdelitev zakasnitve v selekciji posamezne rešitve, ki jo algoritem ovrednoti. Le-ta je enaka številu potencialnih sprememb populacije med vrednotenjem opazovane rešitve, torej številu opravljenih selekcij v času njenega vrednotenja. Večja kot je zakasnitev v selekciji, z večjo zakasnitvijo se dobre rešitve vključujejo v proces tvorjenja novih rešitev in tem slabše algoritem konvergira. Najpomembnejša lastnost porazdelitve zakasnitve v selekciji je njena srednja vrednost (povprečje čez vse rešitve ovrednotene med izvajanjem algoritma), Le-ta se veča linearno glede na število podrejenih procesov in velikost čakalne vrste na podrejenih procesih. Algoritem AMS-DEMO zato najbolj učinkovit deluje na majhnem številu procesorjev in s kratkimi čakalnimi vrstami.

Poskusi in rezultati

Hitrost konvergence algoritma preverimo eksperimentalno, na dveh praktičnih primerih večkriterijskih optimizacijskih problemov. Prvi je uglaševanje parametrov industrijskega procesa kontinuiranega ulivanja jekla, kjer jeklarna stremi po optimizaciji več kriterijev. Gre za nastavljanje intenzivnosti hlajenja jekla na različnih točkah v procesu, ki vpliva na kakovost jekla, porabo hladilne tekočine in varnost proizvodnega postopka [30, 25].

Drugi je testiranje zmogljivosti simulatorja EKGjev. Za kardiologijo in z njo povezane veje medicine je pomembno, da lahko čim bolje simuliramo delovanje srčne mišice [68, 27, 52, 53]. Zgradili smo simulator EKGjev, ki deluje na podlagi modela akcijskega potenciala, z nekaj neznanimi parametri, katerih optimizacija se izkaže kot zelo primerna za reševanje z MOEA [24]. Določanje parametrov namreč lahko poteka preko optimizacije na osnovi simulatorja, kjer poskušamo rezultate simulatorja čim bolj približati izmerjenim EKGjem [24, 37]. Z večkriterijsko optimizacijo lahko raziskujemo tudi zmogljivost simulatorja EKG, tako da opazujemo, katerim značilnostim realnih EKGjev simulator ni zmožen zadostiti.

Za primerjavo z algoritmom AMS-DEMO smo razvili paralelni algoritem imenovan *generacijski DEMO*, ki izvirni DEMO najprej spremeni v generacijski algoritem, nato pa ga paralelizira po sinhronem (običajnem) principu nadrejeni-podrejeni. Ta algoritem pred-

stavlja enostavno paralelizacijo izvirnega algoritma, kakršne so zelo pogoste [46, 51, 38], in je zelo zmogljiv na računalniških arhitekturah, ki mu ustrezajo. Algoritem AMS-DEMO po pričakovanjih konvergira počasneje od algoritmov DEMO in generacijski DEMO. Razlika narašča s številom procesorjev, od statistično nesignifikantne pri majhnem številu procesorjev glede na velikost populacije, do zelo opazne, ko število procesorjev doseže isti velikostni razred kot velikost populacije. Velika prednost algoritma AMS-DEMO in na splošno asinhronega izvedbe paralelizacije po načinu nadrejeni-podrejeni je, da lahko število procesorjev preseže velikost populacije. V poskusih smo simulirali števila procesorjev 10 do 20-krat večja od velikosti populacije in ugotovili, da AMS-DEMO v takih pogojih konvergira slabše, a kljub najde dobre rešitve hitreje, kot če bi se izvajal na manjšem številu procesorjev. Pohitritev namreč ob vsakem dodatnem procesorju naraste in AMS-DEMO lahko v nasprotju z algoritmom generacijski DEMO doseže pohitritve, ki so veliko višje od velikosti populacije.

Podamo tudi enačbi za analitično določanje časa izvajanja, preko katerih lahko primerjamo algoritma AMS-DEMO in generacijski DEMO. Enačbi sta poenostavljeni in upoštevata le najpomembnejše faktorje, ki vplivajo na čas izvajanja. Odločitve za poenostavitve so podprte z merjenji časa posameznih delov algoritmov na testnih problemih, iz katerih se jasno vidi, kateri deli pomembno prispevajo k času izvajanja, in kateri deli se izvedejo v zanemarljivem času.

Kot zadnje opravimo poskuse na heterogeni računalniški arhitekturi, ki kažejo prilagodljivost algoritma AMS-DEMO, saj lahko na takih arhitekturah dobro izkoristi vse procesorje, v nasprotju z generacijskimi algoritmi, kakršen je generacijski DEMO. Pri slednjem bi bili procesorji v heterogeni arhitekturi slabo izkoriščeni – velik delež časa bi porabili za čakanje. Prednosti AMS-DEMO bi se pokazale tudi v primerih delno obremenjenih računalniških sistemov in na problemih, pri katerih vrednotenje rešitev lahko traja različno dolgo. Zato je prilagodljivost algoritma AMS-DEMO, ki jo kaže demonstracijski test na heterogeni arhitekturi, še toliko bolj pomembna.

Prispevki k znanosti

Znanstveni relevantnost disertacije opredeljujejo naslednji prispevki:

- Razvoj novega vzporednega evolucijskega algoritma za večkriterijsko optimizacijo AMS-DEMO. Na podlagi izvirnega zaporednega evolucijskega algoritma s stabilnim stanjem je razvit vzporedni algoritem, ki se od svojega predhodnika poleg zmožnosti izvajanja na raznolikih večprocesorskih arhitekturah razlikuje tudi v podrobnostih izvedbe. Uporabljena je paralelizacija po metodi asinhroni nadrejeni-podrejeni, ki še ni bila sistematično raziskana za uporabo v evolucijskih algoritmih za večkriterijsko optimizacijo.

- Podrobna analiza lastnosti razvitega algoritma AMS-DEMO. Analiza obsega določanje relativne pohitritve algoritma, hitrosti konvergence, predvidevanje časa izvajanja, izkoriščenost procesorjev in sprememb, nastalih zaradi paralelizacije. Razviti vzporedni algoritem primerjamo z enostavnim vzporednim algoritmom generacijski DEMO in izvirnim zaporednim algoritmom DEMO.

- Identifikacija lastnosti vzporednih algoritmov paraleliziranih po načinu nadrejeni-podrejeni z asinhrono komunikacijo. Lastnost poimenujemo zakasnitev v selekciji (angl. selection lag), in jo analiziramo teoretično ter na poskusih opravljenih z algoritmom AMS-DEMO.

- Povečanje zmožnosti računalniško podprtega optimiranja procesnih parametrov v kontinuiranem ulivanju jekla z vidika uporabe računsko zahtevnejših in s tem natančnejših simulatorjev. S tem se poveča uporabnost metodologije v praksi in z njo doseženi prihranki.

- Doseganje višje stopnje ujemanja med računalniško simuliranimi in izmerjenimi EKGji, kot ga omogočajo sedanje metode optimizacije. To bo prispevalo k boljšemu razumevanju delovanja srca in omogočilo preverjanje novih znanstvenih hipotez v kardiologiji.

Zaključki in nadaljnje delo

Algoritem AMS-DEMO, ki je razvit, testiran in analiziran v disertaciji, izpolnjuje skoraj vsa pričakovanja. Čeprav na računalniških arhitekturah in problemih, prilagojenih generacijskim algoritmom, ne izboljša rezultatov veliko enostavnejšega algoritma generacijski DEMO, se veliko bolje izkaže v drugih okoljih. Izkaže se z robustnostjo, veliko prilagodljivostjo, zaradi katere zmore učinkovito delovati na heterogenih računalniških arhitekturah, delno obremenjenih procesorjih in na problemih, kjer trajanje vrednotenja rešitev ni konstantno. Dobrodošla je tudi nenačrtovana lastnost, da se lahko izvaja na številu procesorjev, ki je večje od velikosti populacije.

V nadaljnjih raziskavah bomo algoritem AMS-DEMO preiskusili na dodatnih problemih. Ker je učinkovitost algoritma na vzporednih računalniških arhitekturah povezana predvsem z načinom paralelizacije in manj z izvirnim algoritmom DEMO, bi bil naslednji korak lahko tudi posplošenje paralelizacijske metode in njena izvedba na podobnih evolucijskih algoritmih, tako eno- kot večkriterijskih. Nazadnje bi bila dobrodošla tudi globlja analiza zakasnitve v selekciji, kot osnovne lastnosti paralelizacije po asinhroni metodi nadrejeni-podrejeni.

List of Symbols and Abbreviations

Abbreviation	Description	Definition
AMS-DEMO	asynchronous master-slave DEMO	page 45
EA	evolutionary algorithm	page 18
ECG	electrocardiogram	page 33
DE	differential evolution	page 20
DEMO	differential evolution for multiobjective optimization	page 21
MO	multiobjective optimization	page 16
MOEA	multiobjective evolutionary algorithm	page 20

Chapter 1

Introduction

In this chapter, motivation for the dissertation is first presented, followed by its goals. Then the scientific contributions are listed, and finally, the overview of the dissertation is given.

1.1 Motivation

Real-life optimization problems are frequently very complex, making analytical optimization infeasible. If the problem can be modeled, numerical optimization can be used instead. Models of complex problems usually include a computer simulation of the problem, which, given a potential solution to the problem, can be used to evaluate how well the solution solves the problem.

To solve multimodal optimization problems, heuristic methods can be used; for example, evolutionary algorithms. Although very powerful, heuristic methods have a weakness – a demand for evaluation of a large number of potential solutions. As real-life problems are often represented by complex simulators, this makes optimization with heuristic methods time consuming. On a positive note, the evaluations of solutions are largely independent of each other – making heuristic methods somewhat implicitly parallel. With some effort, this can be exploited by parallelizing the methods and running them on parallel computers, this is computers with multiple central processing units or processors, as we shall refer to them from now on.

In case of evolutionary algorithms, there are several well known parallelization principles, that either exploit the implicit parallelism directly, or rearrange the algorithms in a way that provides additional parallelization options. The traditional parallelization principles, however, have their limits. One is the population size in population based algorithms, which almost without exception defines the upper limit on the number of processors that can be utilized with the parallel algorithm. With an ever increasing numbers of processors in computers ranging from the personal computers to state of the art supercomputers, this limit is becoming increasingly important. Another is the require-

ment for frequent synchronization between the collaborating processors, which stems a requirement for load balancing the processors and is a great obstacle to efficient use of parallel computers comprising heterogeneous processors.

Furthermore, extending the single-objective optimization into a growingly popular multiobjective optimization, which is a newer concept albeit a more general one, complicates the parallelization of evolutionary algorithms somewhat, mostly because of the greater demand for synchronization. Therefore, algorithms that can run asynchronously should have an advantage over those that can not.

1.2 Goals

We seek ways to make a parallel evolutionary algorithm for multiobjective optimization that is to a large extent independent of the hardware used to run it and of the problem that it is intended to solve. This means it should be able to run efficiently on sets of processing units, on any number of them, and have no requirements on the network topology.

There are two goals of this dissertation. First is to present AMS-DEMO, a parallel algorithm for multiobjective optimization of numerical problems. AMS-DEMO has been developed by parallelizing DEMO, a purely serial algorithm. It uses a version of asynchronous master-slave parallelization method, which is not entirely novel but can be rarely found in the literature. AMS-DEMO therefore also serves as a demonstration of this parallelization method. We will present the types of problems best suited for solving with AMS-DEMO and contrast them to the types of problems that are ill suited. We will also compare AMS-DEMO to a simpler parallel algorithm, called generational DEMO, which works equally well as AMS-DEMO in some specific cases, and could therefore be preferred in those cases on the account of its simplicity.

The second goal is to present a detailed analysis of AMS-DEMO. We accomplish it by experimentally testing the algorithm on two real-life problems. The results of the tests are then compared to the results obtained by the original DEMO and generational DEMO. Besides giving a clear picture of the efficiency of AMS-DEMO on the tested problems, experimental tests and their results can also be used to predict the behavior of AMS-DEMO on similar problems and the behavior of the asynchronous master-slave method if it were used to parallelize a similar algorithm. Additionally, we present a theoretical analysis of the changes made to DEMO algorithm for the development of AMS-DEMO, identify a key property that has emerged from the parallelization – selection lag, and discuss the ways in which it influences the algorithm execution time. A way to calculate mean selection lag from hardware and algorithm parameters is also given.

1.3 Scientific Contributions

This dissertation makes the following scientific contributions:

- Development of AMS-DEMO – a new evolutionary algorithm for multiobjective optimization that is able to run efficiently on a heterogeneous set of processors. A rare parallelization type – an asynchronous master-slave parallelization – is used to transform the original DEMO to parallel AMS-DEMO.
- Analysis of AMS-DEMO based on tests made on two real-life problems and on varying number of processors, including very high numbers of processors, which were emulated. Analysis includes run times, convergence rates, relative speedups, utilization of multiple processors, and changes to the algorithm because of the parallelization. AMS-DEMO is compared against the original DEMO and generational DEMO.
- Identification of a key property defining the behavior of AMS-DEMO as well as any other algorithm parallelized with the asynchronous master-slave type. We name this property selection lag, assign it the symbol l, and analyze it analytically as well as experimentally on the test problems.
- Allowing additional options in computer-assisted optimization of process parameters of continuous steel casting, mostly by allowing more complex simulators with greater fidelity of simulations. The usability of this optimization is thus increased, along with the prospective savings arising from it.
- Achieving greater fidelity in simulation of ECGs than allowed by current simulator-based optimization methods. Consequently, working towards better understanding of the electrical activity of the human heart and opening new directions in cardiology research.

1.4 Overview of the Dissertation

Dissertation is further organized as follows. Chapter 2 presents backgrounds of multiobjective optimization, evolutionary algorithms, algorithm parallelization, and the ways of combining these concepts into parallel evolutionary algorithms for multiobjective optimization. Chapter 3 presents two multiobjective optimization problems that are later used in the experimental evaluation of the presented algorithm. In chapter 4, two parallel implementations of DEMO are presented. The first one is straight-forward to implement and is contrasted against the second one – AMS-DEMO, which is the main contribution of the dissertation. All the important implementation details of AMS-DEMO are also explained and the selection lag is identified in the same chapter, as an algorithm property

which defines its behavior in dependence of the computer architecture and some problem properties. Chapter 5 describes the experiments made on the presented multiobjective optimization problems for the evaluation of AMS-DEMO and their results. The results are grouped into three categories. The results of optimization, the convergence, and the speedup of both parallel algorithms. The chapter ends with theoretical analysis of AMS-DEMO – supported by the measured results – for the prediction of AMS-DEMO behavior on various computer architectures and problems. Chapter 6 concludes the dissertation and gives suggestions for further work.

Chapter 2

Background

This chapter presents the background knowledge necessary for understanding the proposed algorithm and its analysis. It starts with optimization, extends it into multiobjective optimization, followed by a popular stochastic optimization method – evolutionary algorithms. Then parallel computer architectures are presented. The chapter ends with the description of the parallelization of evolutionary algorithms where the concept of speedup is also introduced.

2.1 Optimization

Optimization is a tool for finding either minimum or maximum of a selected property of a given system. It starts with the identification of an *objective*, that is, a measure of the property in question, which can be quantified by a single number. The value of the objective depends on the *parameters* (also called *decision variables*) of the given system. The task of the optimization is to find values of the parameters that optimize - either minimize or maximize - the objective. The parameters are usually constrained, most often by having at least a lower and upper limit, but more generally, having any number of equality and inequality constraints imposed on either single parameters or sets of them.

Formally, an optimization problem can be defined as a task that requires optimizing the objective function (also called cost function) f:

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

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

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

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

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

and equality constrains

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

are called feasible solutions.

2.2 Multiobjective Ooptimization

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, which 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

$$\mathbf{f}(\mathbf{x}) = [f_1(\mathbf{x}), f_2(\mathbf{x}), ..., f_m(\mathbf{x})]^{\mathrm{T}}. \tag{2.1}$$

There are two Euclidean spaces associated with multiobjective optimization – 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 by the *Pareto dominance* (named after Vilfredo Pareto (1848–1923), an Italian economist, sociologist, and a pioneer in the field of multiobjective optimization). Given two objective vectors, $\mathbf{a}$ and $\mathbf{b}$, $\mathbf{a}$ is said to *dominate* $\mathbf{b}$ ($\mathbf{a} \prec \mathbf{b}$) if and only if $\mathbf{a}$ is better than $\mathbf{b}$ in at least one objective and is not worse than $\mathbf{b}$ in all other objectives. Formally, assuming all objectives are to be minimized:

$$\begin{aligned} \mathbf{a} \prec \mathbf{b} \quad & \text{iff} \\ \forall k \in \{1, 2, \ldots, m\} : \mathbf{a}_k \leq \mathbf{b}_k \quad & \text{and} \\ \exists l \in \{1, 2, \ldots, m\} : \mathbf{a}_l < \mathbf{b}_l & \end{aligned} \tag{2.2}$$

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. Figure 2.1 shows five solutions to this problem in the objective space. Comparing solution $\mathbf{a}$ with other solutions, we can observe that $\mathbf{a}$ dominates $\mathbf{b}$ since it is better than $\mathbf{b}$ 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 $\mathbf{c}$ as it is better than $\mathbf{c}$ in objective f_2 and not worse in objective f_1. On the other hand, $\mathbf{d}$ outperforms $\mathbf{a}$ in both objectives, therefore $\mathbf{d}$ dominates $\mathbf{a}$ or, in other words, $\mathbf{a}$ is dominated by $\mathbf{d}$. However, regarding $\mathbf{a}$ and $\mathbf{e}$, 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 $\mathbf{a}$ and $\mathbf{e}$ are incomparable.

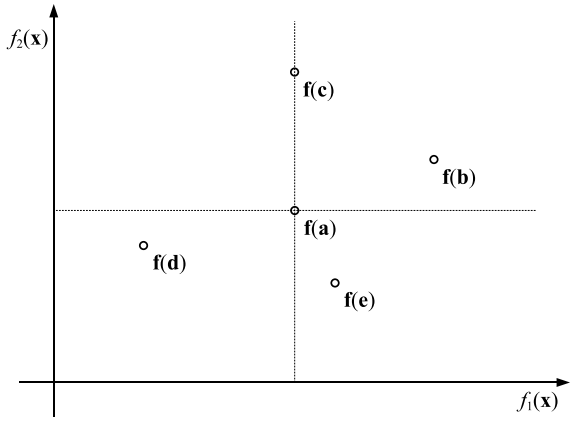

Figure 2.1: Comparison of solutions to a multiobjective optimization problem in the objective space.

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 Figure 2.1). 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 Figure 2.2 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 nondominated front of the entire feasible search space the *Pareto optimal front*.

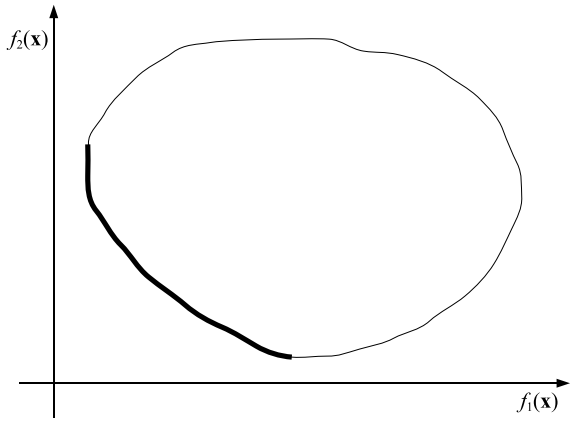

Figure 2.2: Nondominated front of solutions in the objective space (both objectives need to be minimized).

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 be as diverse as possible.

Traditionally, multiobjective optimization was performed by methods that transform problems into single objective form and then solve them using techniques of single objective optimization. We refer to these methods as *classical* methods [19]. Their main difficulties are the inability to produce more than one Pareto-optimal solution per run, the inability to find all Pareto-optimal solutions in non-convex problems, and the requirement for some problem knowledge. Their practical use is mostly limited by the requirement to run them several times to produce multiple Pareto-optimal solutions, which, in contrast, is not required by population-based methods. From these, the most commonly used are evolutionary algorithms, which we will discuss next.

2.3 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 [28, 17]. 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 2.1, starts with a population of randomly created individuals, and iteratively improves them by employing evolutionary mechanisms, such as survival of the fittest and exchange of genetic information between the individuals. The iterative steps are called generations, and in each generation the population members undergo fitness evaluation, selection, and variation. Note that we shall refer to the fitness evaluation as evaluation from now on.

The selection phase of the algorithm is an artificial realization of the Darwinian principle of survival of the fittest. 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 to create new candidate solutions to the considered problem. For this purpose, the EA applies variation operators, such as *crossover* and *mutation*, to the individuals. The crossover operator exchanges randomly selected

Algorithm 2.1: Evolutionary Algorithm (EA)

```
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          select two parents from ℙ
7          create two offspring by crossing the parents
8          mutate the offspring
9          evaluate the offspring
10         add the offspring into ℙ_new
11     until ℙ_new is full
12     copy ℙ_new into ℙ
```

components between pairs of individuals (*parents*), while the mutation operator 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 mechanism for 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.

EAs can be divided into *generational* model and *steady-state* model algorithms. In the generational model, each generation begins with a population of μ solutions, from which λ offspring are created by the application of variation operators, where usually $\lambda \geq \mu$. Then, the fitness of the offspring is evaluated and in the selection phase, μ solutions are selected to form the population of the next generation, either from both the population and the offspring, called $(\mu + \lambda)$ selection, or from only the offspring in (μ, λ) selection. In the end, the whole population is replaced by the selected solutions.

The steady-state model in contrast, does not replace the whole population in a single

step. A single step rather consists of a single application of variation operators, usually creating one or two offspring that is/are compared to the same number of solutions from the population in the selection phase and may replace them immediately. Steady-state model corresponds to $(\mu + \lambda)$ generational selection model, where λ is the number of offspring created by a single application of variation operators.

2.3.1 Differential Evolution

A somewhat specialized EA is *Differential Evolution* (DE) [49, 48]. It was designed for 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 2.2, new candidates are constructed through operations such as vector addition and scalar multiplication (in line 7, 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 2.2: 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      foreach p_i ∈ ℙ do
6          randomly select three different solutions s_1, s_2, s_3 from ℙ
7          create a candidate solution c ← s_1 + F · (s_2 − s_3)
8          alter c by crossover with p_i
9          evaluate c
10         if c is better than p_i then
11             add c into ℙ_new
12         else
13             add p_i into ℙ_new
14     ℙ ← ℙ_new
```

2.3.2 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 Algorithms* (MOEAs) [19, 16, 3].

The two most well known MOEAs are NSGA-II and SPEA2, which are also often used for evaluation of other MOEAs. We give short overview of both methods.

Nondominated sorting evolutionary genetic algorithm II (NSGA-II) [20] is an improvement of an earlier algorithm NSGA. It is based on forming a new population from the best individuals of the current population and its offspring, which are selected using a fast sorting of nondominated solutions coupled with crowding distance sorting. Crowding distance sorting is a parameter-less niching approach to diversity preservation. NSGA-II is a generational algorithm that implements elitism and can be used for problems with continuous (real-valued) parameters, as well as discrete parameters. Its main strength is good performance on large populations, because its time complexity is only $O(Mn^2)$, where M is the number of objectives and n is the population size.

Strength Pareto evolutionary algorithm 2 (SPEA2) [72] is an improvement of on the already successful algorithm SPEA. It is named after its way of assigning fitness to solutions – as a sum of strength values of all the solutions that dominate the observed solution (note that fitness value is to be minimized in SPEA2). Strength of a solution equals the number of solutions the observed solution dominates. In addition, density information based on the k-th nearest neighbors method [57] is used to sort individuals that would otherwise have identical fitness values. SPEA2 is a generational algorithm and combines the current population with its offspring to form a new generation. It also implements elitism through an archive of best nondominated solutions of the population. Mating is implemented as tournament selection with replacement.

Next section presents DEMO, the multiobjective optimization algorithm used as a base for our parallel multiobjective optimization algorithm.

2.3.3 Differential Evolution for Multiobjective Optimization (DEMO)

Based on the single-objective DE is *Differential Evolution for Multiobjective Optimization* (DEMO) [55, 66]. It extends DE into multiobjective optimization algorithm by changing the mechanism for deciding which solutions to keep in the population (see Algorithm 2.3). DE implements a straight-forward selection between the parent and its offspring, which keeps the better of the two and discards the worse, based on the comparison of fitnesses. In DEMO, comparison of fitnesses has to be extended because of the multiple objectives. Pareto dominance relation is used, which allows vector comparisons but does not always mark one of the compared solutions as better than the other. If one dominates the other, than it is kept while the other is discarded, but if the two are incomparable – neither

one dominates the other – than both are kept. In the latter case, the population size increases by one, which is unwanted and should be compensated for. Therefore, the population is truncated back to the original population size n once every n solutions have been evaluated. The truncation mechanism uses nondominated sorting and the crowding distance metric in the same manner as in the NSGA-II multiobjective algorithm [20]. In addition, DEMO is converted from generational to steady-state by applying selection to current population instead of putting its output (the surviving solutions) into a new population which would later entirely replace the current population, as done in DE. Pseudo code of DEMO is shown in Algorithm 2.3.

Algorithm 2.3: 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      foreach p_i ∈ ℙ do
5          randomly select three different solutions s_1, s_2, s_3 from ℙ
6          create a candidate solution c ← s_1+F·(s_2 − s_3)
7          alter c by crossover with p_i
8          evaluate c
9          if c ≺ p_i then
10             replace p_i with c in ℙ
11         else if not p_i ≺ c then
12             add c into ℙ
13     if ℙ contains more than popSize solutions then
14         truncate ℙ
```

DEMO as presented so far is only one of the possible DEMO variants, and is called DEMO/parent on the account of the selection acting on the candidate solution and its parent. There are two additional DEMO variants – DEMO/closest/obj and DEMO/closest/dec, where the selection acts on the candidate solution and the most similar solution from the population, respectively. Similarity is defined as the Euclidean distance between pairs of solutions either in objective or decision space. Since the latter two variants are more computationally expensive than DEMO/parent but do not bring any important advantage over it [55], we use DEMO/parent as the base for parallelization.

2.4 Parallel Computer Architectures

Traditional serial computers are referred to as *single instruction, single data* (SISD) computers [40] according to Flynn's taxonomy [32]. Such computers are only able to execute a single instruction on data from a single location in memory. According to Flynn's taxonmy, there are two ways of extending serial computers into parallel computers – computers that are able to do several tasks concurrently. The first one is by adding the ability to process multiple data using the same instruction stream, creating *single instruction, multiple data* (SIMD) computers. A good example are GPUs (graphic processing units) which render graphics by executing the same commands on multiple pixels in parallel. The second, orthogonal way of adding parallelism is by adding the ability to execute multiple instructions in parallel. Computers that do so on the same data stream are called *multiple instruction, single data* (MISD) computers, but are very rare because the requirement to process the same data using multiple different instruction streams is also very rare. On the other hand, *multiple instruction, multiple data* (MIMD) computers that allow concurrent processing of multiple data streams, each one using its own instruction stream, are very common. Such computers comprise several processing units, which can be either placed on the same integrated circuit, or on different integrated circuits with shared memory, or even on completely remote locations, each with its own memory. Although MIMD computers have the highest requirement for hardware resources, they are very simple to implement, since they can be assembled by stacking together off-the-shelf SISD computers and connecting them via network. They are also the most versatile and can handle all the tasks that the other groups of computers from Flynn's taxonomy can.

Multiple processors do not make a MIMD computer without some form of interaction between the processors. There are two means of processors interaction. *Shared address space* is the first one, and implements interaction by allowing different processors read and write access to the same memory address space. Although it is mostly used on computers which share memory among the processors (*shared memory* architecture), it can also be implemented on computers which provide separate physical memory to different processors by appropriate hardware mapping of memory addresses. *Message passing* is the second means of processor interaction, and allows processors to interact only by passing messages to each other. Each processor of the system has its own local memory that is not accessible to other processors and can not be used to share data. Message passing architectures have an advantage of being less complex and less expensive than shared address space architectures, which on the other hand offer easier interface to interaction, greater flexibility in programming, and faster access to shared data.

2.5 Parallelization of EAs

Fitness evaluation is often computationally expensive, especially when performed via simulation, making the optimization impractical because of the long execution time it requires for the calculation of a large number of evaluations. It is therefore beneficial to parallelize the algorithm to be used on multiple processors and thus shorten the execution time. The parallelization seeks to modify the algorithm in a way that maximizes speedup [4] – the factor of how much shorter the execution time is on multiple processors than on a single processor. EAs are an example of inherently parallel algorithms because they work on a population of solutions, which allows for an efficient use of multiple processors in parallel. This means that EAs may be easily parallelized and large speedups may be achieved.

MOEAs are a subclass of EAs and can be parallelized in one of the four EA parallelization types [12, 7, 67, 42]; three basic: *master-slave* (also called *global parallelization*), *island model*, *diffusion model* (also known as *cellular model*); and *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 their inherent parallelism. Consequently, they traverse the search space identically to their serial counterparts. They can be visualized as a master processor running a serial EA with a modification in creation and evaluation of solutions. Instead of creating and evaluating solutions serially, one at a time, until the entire population is evaluated, solutions are created and evaluated on the master and slave processors in parallel. This, however, does not apply to steady-state algorithms, in which the creation and evaluation of a single solution depends on the evaluation result of the previously generated solution. 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 [4] (speedup that is close to the upper theoretical limit) is possible. Master-slave parallelization is popular with MOEAs, ranging from very simple implementations [46] where the master runs on a separate processor, and [51] where the master processor also runs one slave (usually as two separate processes). There are also implementations for heterogeneous computer architectures, where load-balancing has to be implemented. Examples are [38] with pool-of-tasks load balancing algorithm, and [59, 34] with an asynchronous master-slave parallelization of a steady-state algorithm where the load balancing is implicit.

Island model EAs, in contrast, are multiple-population algorithms, consisting of several largely independent subpopulations that occasionally exchange a few solutions. In

an island EA, each processor represents an island, running a serial EA on a subpopulation. A new operator is introduced – migration, that handles the exchange of solutions 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. Because the communication is less frequent, its overhead is 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. Heterogeneous computers can also be very efficiently used by the island model EAs, as shown in [5]. MOEAs may use the island model as defined for EAs in general or extend it by dividing multiobjective optimization into subproblems and then assigning each island a different subproblem. One approach to this is shown in [47], where each subpopulation (an island) is assigned a fitness based on a different objective function. Another approach is dividing the objective space into segments and then exploring one segment per island. The difficulty of this approach lies in guiding or bounding the islands to search within their segment, and in dividing the objective space fairly among the islands without the prior knowledge of its shape. The attempts so far seem promising [22, 11, 60], showing good speedups on small number of islands.

Diffusion model EAs split the population into multiple small subpopulations and divide them among the processing nodes. Every subpopulation is allowed to communicate only with a predefined neighborhood of other subpopulations – the variation operators are only applied on sets of solutions from one subpopulation and its neighborhood. Diffusion model EAs can also be considered single-population with structurally constrained interactions between solutions. Parallelization of this type has the largest communication overhead among the mentioned types and requires computer architectures with numerous processors and fast interconnections. Speedup and efficiency depend greatly on the properties of interconnections and the suitability of the problem to the structural constraints imposed by the algorithm. An example of diffusion model MOEA can be found in [56].

Hybrid parallel EAs are an attempt to minimize the weaknesses of the basic algorithm types through their hierarchic composition. For example, the island model may be implemented on top of the master-slave, providing possibility to use all available processing nodes, while keeping the number of islands variable. Hybrid EAs can be adapted to the underlying hardware architecture to a high degree, but their design and implementation are more complex. The combination of master-slave and island models is most often used in hybrid parallel EAs and MOEAs, as for example in [47, 34].

2.5.1 Taxonomy of Parallel Metaheuristics for MO

MOEAs are representatives of the parallel metaheuristics for multiobjective optimization, with two main parallelization strategies used – single-walk and multiple-walk [45].

Single-walk parallel metaheuristics are aimed at speeding up the underlying sequential algorithms while preserving their basic behavior, which means that the sequential traversal through the search space is preserved by applying parallelism in two ways – by *parallel function evaluation* – fitness function is computed for several solutions in parallel, or by *parallel operator* – the search operators of the method are applied in parallel. Master slave parallelization type is the implementation of the single-walk parallel function evaluation strategy, because fitness function is most often the most complex and time demanding step of the EA, while it can be calculated concurrently for multiple solutions.

Multiple-walk parallel EAs represent an effort to improve the solution quality by means of multiple concurrent traversals (search threads) through the search space. Search threads may either be completely independent or cooperative (they share the collected information among themselves). Depending on the way the Pareto front is built during the optimization process, two variants may be considered. In algorithms with *centralized Pareto front*, search threads are continuously improving a global Pareto front throughout the optimization process. In contrast, algorithms with *distributed Pareto front* implement separate local Pareto fronts for each search thread, and a mechanism that merges local fronts into a global front at the end of the optimization procedure. Updating a global Pareto front is a communication intensive task and for performance reasons the algorithms with centralized Pareto fronts without exception implement phases during which they work on distributed Pareto fronts and update the global Pareto front only at the end of each phase [45]. Island and diffusion models are examples of the multiple-walk strategy and may use either distributed or centralized Pareto front.

2.5.2 Speedup

Speedup is a measure that captures the relative benefit of solving a problem in parallel [40]. Formally, speedup S on p processors is the ratio of the time required to solve a problem on a single processor $t(1)$ to the time required to solve it on p identical processors $t(p)$:

$$S(p) = \frac{t(1)}{t(p)} \, . \tag{2.3}$$

Traditionally, speedup of an algorithm is defined relative to the best known serial algorithm, making $t(p)$ the time required by the parallel algorithm that we wish to calculate speedup for, and $t(1)$ the time required by the best known serial algorithm. We will call thus defined speedup *absolute* to separate it from the *relative speedup*, which is defined relative to the original serial algorithm. We feel that relative speedup is more meaningful in this dissertation and will use it exclusively, referring to it simply as speedup.

Calculating absolute speedup is problematic for two reasons. The first one is the definition of the best algorithm for the problem – is this the best optimization algorithm overall, or simply the best optimization algorithm for the given problem? The former definition is too broad; there is no absolutely best optimization algorithm known, nor is there a way of a meaningful comparison of optimization algorithms that would yield only one that is absolutely best. The latter definition lacks generalization and is meaningless to us, because we are trying to build a general parallel optimization algorithm, and are not only interested in solving our test problems in record breaking time. The second reason for the absolute speedup being problematic is that it contains no information on the properties of the parallelization method, which we are trying to evaluate along with the proposed parallel algorithm. Relative speedup, on the other, is less dependent on the optimization problem we are trying to solve and tells us mostly how efficient the parallelization method is. It can also be extended to absolute speedup or speedup relative to any other algorithm one might wish to compare our proposed parallel algorithm to – by multiplying it with the speedup of the original algorithm relative to the desired baseline algorithm.

The upper limit for speedup equals p and is called *linear speedup*. Sometimes algorithm implementations seem to go over this limit, yielding *super linear speedup* – speedup higher than p. There are two possible reasons for super linear speedup. The first reason are the additional resources provided by the parallel architecture, such as increased cache size and memory throughput. These can have big enough effect on some algorithms to push their speedup above p, but are very platform specific. Super linear speedup is therefore caused by the specific computer architecture and not by the algorithm itself. The second reason lies in the changes made to the algorithm while parallelizing it. Sometimes changes that are aimed at increasing the parallel portion of the algorithm for a more effective parallelization inadvertently improve the algorithm. An excellent example are the island model EAs. Often a serial single-population EA is parallelized into an island model EA, making it multiple-population. Although the obtained parallel algorithm can exhibit speedups much larger than p over the original algorithm, this only indicates the inefficiency of the original algorithm. Using a multiple-population serial algorithm for a comparison instead of the original algorithm, the obtained parallel algorithm would only exhibit speedup equal to or lower than p.

A measure related to speedup is *efficiency*, which is simply the speedup normalized with the number of processors:

$$E(p) = \frac{S(p)}{p} \ . \tag{2.4}$$

Efficiency thus lies between 0 and 1, with 1 corresponding to the linear speedup. Increasing the number of processors is accompanied with an increase in communication overhead, causing the efficiency to drop. Some parallel systems – a combination of an architecture

and an algorithm – are *scalable*, i.e. have the ability to keep the efficiency fixed if both the problem size and the number of processors increase simultaneously.

In case of the master-slave EAs, speedup is easy to measure because these algorithms traverse the search space identically to their serial counterparts. Therefore, using the same settings for both, they can be compared directly. More care should be taken when dealing with parallel EAs that do not traverse the search space in the same way as their serial counterparts. Basic island and diffusion EAs that have been developed from single population serial EAs, are such examples. These algorithms sometimes exhibit super linear speedup and could be serialized, producing a new serial algorithm that is faster than the original one. Only this new serial algorithm should then be used as a base for calculating the speedup. Furthermore, the new serial algorithm would have some additional parameters that are limited by the hardware on the parallel version – such as the number of islands or the structure of interconnections. When comparing the serial and parallel version, these parameters should be set to their best values and not identically to the parallel version. The only limiting factor for serialization could be hardware (for example, multiple-population EAs require more memory than single-population EAs). In such cases, parallelization would alleviate hardware constraints as well as provide the speedup. The obtained speedup would be due to two factors, the parallel execution and the algorithm improvements, with either factor unobtainable from the measurements alone.

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

$$S_{max} = \frac{1}{(1-P) + \frac{P}{p}} \ , \tag{2.5}$$

where P is the parallel portion of the algorithm and p is the number of processors. The actual speedup of an algorithm will depend on how well the parallel portion can be spread up among p 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 communication between processors is taken into consideration, P can still reach very high values if fitness evaluation is complex and time consuming. On the other hand, p is limited by the population size N. 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:

$$\lim_{P \to 1} S_{max} = \lim_{P \to 1} \frac{1}{(1-P) + \frac{P}{N}} \ = \ N \ . \tag{2.6}$$

Another important observation is that not only should $N \leq p$, but N should also divide p (we write this as $N \mid p$), for the algorithm to fully utilize all processors. The algorithm requires $\lceil \frac{N}{p} \rceil$ iterations to fully evaluate the population and therefore has $\lceil \frac{N}{p} \rceil p$ processor time slots to fill with N 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 ($N \bmod p$) unallocated slots per generation, leaving the same number of processors idle. Knowing this we can derive the effective number of processors used by the algorithm: $p_{\text{eff}} = N/\lceil \frac{N}{p} \rceil$. For example, having $p = 10$ and $N = 15$, we get $p_{\text{eff}} = 7.5$. We see that having the population size which is not a multiple of number of processors reduces the effective number of processors. We could lower p to 8, the first integer greater than p_{eff}, and still achieve the same speedup. Substituting p with p_{eff} in Equation 2.5, we can write the equation for maximum achievable speedup as

$$S_{\max}(p) = \frac{1}{(1-P) + \frac{P \times \lceil \frac{N}{p} \rceil}{N}} \,. \tag{2.7}$$

An example of $S_{\max}(p)$ for population size $N = 32$ and parallel fraction $P = 1$ is shown in Figure 2.3.

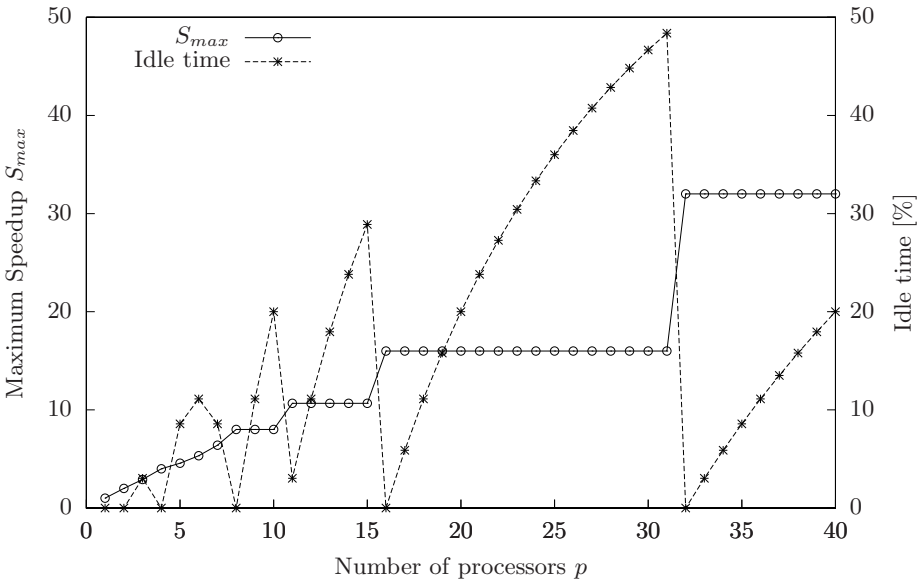

Figure 2.3: Maximum speedup S_{max} and processor idle time vs. the number processors p for a master-slave parallel EA with parallel portion $P \to 1$ and population size $n = 32$.

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, the best population size for a given problem can be only approximately determined. Thus an

approximate interval rather than the exact number for the best size is usually found. Because population size can be chosen as any number from this interval, it can usually be set to a multiple of the number of processors. In cases when optimal selection of the population size within the interval is not possible, the maximum speedup is smaller by the factor S_{opt}, which is the speedup of the EA with the optimal population size relative to the EA with the observed population size:

$$S_{\text{max}}* = \frac{S_{\text{max}}}{S_{\text{opt}}} \ . \tag{2.8}$$

Chapter 3

Examples of Multiobjective Optimization Problems

This chapter presents two real-life multiobjective optimization problems, that were used for AMS-DEMO evaluation. The first problem is taken from industry and deals with tuning process parameters (the amount of cooling in different stages) for steady-state steel casting, which we will also refer to as the *cooling problem*. The second problem comes from theoretical research of human heart electrical activity, where ECG model parameters are required to be estimated from the desired output of the model. We shall refer to it as the *ECG problem*.

3.1 Tuning of Process Parameters for Steady-State Steel Casting

Continuous casting of steel is widely used at modern steel plants to produce various steel semi-manufactures. For the plant to operate safely while producing the highest quality steel, several parameters of the casting process must be properly set. We will present this optimization problem, and add a short description of the simulator and its parameters.

3.1.1 Optimization Problem

The continuous casting process is schematically shown in Figure 3.1. 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 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.

The secondary cooling area of the casting device is divided into nine 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

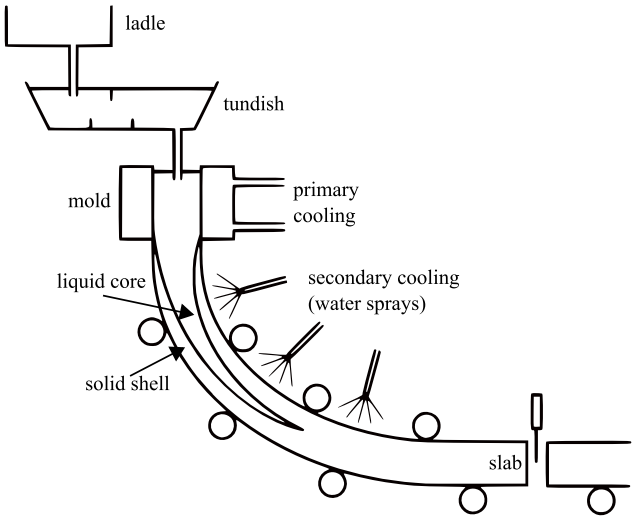

Figure 3.1: A schematic view of continuous casting of steel.

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:

$$f_1 = \sum_{i=1}^{N_Z} |T_i^{\text{center}} - T_i^{\text{center}*}| + \sum_{i=1}^{N_Z} |T_i^{\text{corner}} - T_i^{\text{corner}*}|, \tag{3.1}$$

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.

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 the second objective, f_2:

$$f_2 = |l^{\text{core}} - l^{\text{core}*}|, \tag{3.2}$$

and the optimization task is then to minimize both f_1 and f_2 over numerous 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, as specified in Table 3.1. 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: $f_2 \leq \Delta l^{\text{core}}_{\text{max}}$. Solutions violating the water flow constraints or the core length constraint are considered infeasible. In our experiments, 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.

3.1.2 Mathematical Model

A prerequisite for optimization of the continuous casting 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 Equations (3.1) and (3.2). For this purpose we use a numerical simulator of the process, which considers steady-state conditions, i.e. with parameters held constant in time. The simulator is based on the Finite Element Method (FEM) discretization of the temperature field and the related nonlinear heat transfer equations are solved with relaxation iterative methods [30].

The simulator is set to slab cross-section of 1.70 m $\times$ 0.21 m and casting speed of 1.6 m/min that is 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 are 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. Target temperatures and parameter constraints are shown in Table 3.1.

3.2 Parameter Estimation for the ECG Simulator

Electrocardiogram (ECG) is a diagnostic and monitoring tool that records heart activity by measuring electrical currents, originating in the heart, on the body surface. Modeling the electric activity of a human heart provides useful insight into ECG generating mechanisms that can in turn be used to further the understanding of ECG and improve its diagnostic benefits. There are two main approaches to modeling which attack the problem at different levels. One is modeling of ion currents on the level of individual myocardium (heart muscle tissue) cells as for example in [61]. Due to complexity of realistic cell models, simulations of this sort have very high computational cost and are mostly used to simulate smaller patches of tissue and not the whole heart. Another approach uses simplified cell model consisting of only action potential (AP) – a function defining cell's electric activity – that decreases computational cost by several orders of magnitude and thus enables simulation of the whole heart or a heart cross section [52, 53]. We used

the second approach in [24, 23], where an inverse problem of determining myocardium cell properties from the measured ECG was solved using a newly developed ECG simulator and a simulation-based optimization.

3.2.1 Optimization problem

One important aspect of building an ECG simulator is knowing its limits, i.e. knowing which ECG phenomena can or can not be simulated with it. We are aware of several limitations imposed by the simulator, such as the inability to simulate all of the ECG features because of the prohibitively large spatial resolution it would require; the incorrect absolute ECG amplitudes and even the relations between the amplitudes of ECGs measured at different positions on the body because of the simplified model of the body that assumes uniform and infinite conductivity of the simulated tissues and the air; and the inability to simulate local defects in myocardial tissue because the simulator implements layered myocardium with identical properties of APs in each layer. In addition to these predicted limits, we have identified several additional limits through the simu-

Table 3.1: Target temperatures and water flow constraints for the cooling problem

Zone number	Target [°C]	Parameter number	Min. flow [m^3/h]	Max. flow [m^3/h]
Center positions				
1	1050	1	0	50
2	1040	2	0	50
3	980	3	0	50
4	970	4	0	10
5	960	5	0	10
6	950	6	0	10
7	940	7	0	10
8	930	8	0	10
9	920	9	0	10
Corner positions				
1	880	10	0	50
2	870	11	0	50
3	810	12	0	50
4	800	13	0	10
5	790	14	0	10
6	780	15	0	10
7	770	16	0	10
8	760	17	0	10
9	750	18	0	10

lation of various scenarios. We present here a scenario, which was found to be the most appropriate for evaluation of the parallel algorithms.

One of the advantages of performing ECG simulation on a 3-dimensional heart model is the ability to simulate multiple ECGs on different body locations simultaneously. Similarly, ECGs are recorded at different body locations in real measurements. For this scenario, we simulate ECGs on two locations on the body, that coincide with the locations of V2 and V5 electrodes of the standard 12-channel ECG measurement [43]. This scenario aims to lower the computational complexity of the simulation as much as possible, therefore only a prominent ECG feature called the *T wave* is simulated simultaneously on the two specified electrodes. In real ECG measurements, the location of the T wave peak is delayed by several ten milliseconds on V5 compared to V2. To evaluate to what extent the simulator is able to replicate this delay, we perform a two-objective optimization, where objectives are the Pearson correlation coefficients between the simulated and measured ECG on V2 and V5.

3.2.2 Computer model

We use an improved ECG simulator, based on the one described in [24], which was successfully used to demonstrate multiple ways of generating U wave, which is one of the smallest features of ECGs. The presented simulator has been improved in three areas – the heart model shape, positioning of observation points (locations on the body where the ECG is simulated) and the AP model.

Ultimately the aim is to use medical scans for the shape of the heart model, and either a mathematical model for layering, or an improvement of the simulator that would eliminate the requirement for layers. Currently we continue using the mathematical model for simultaneous creation of shape and layers. We are able to create heart models comprising two ventricles of independent sizes and independent numbers of layers. A model of the heart, comprising of 241135 cubic millimeter cells with left ventricle representing just over 80 % of total mass, is presented in Figure 3.2. Because we are simulating only one of the most prominent features of the ECG in this scenario, we do not need the full resolution and complexity of the presented model. Therefore, we double the size of model cells in all dimensions, increasing their volume 8 times, from 1 mm^3 to 8 mm^3. This way we get a coarse model which consists of 8 times less cells, allowing for approximately 8 times faster simulation. In spite of lower spatial resolution, ECGs simulated using the coarse model are very similar to the ones generated by the basic model. The differences in T waves – the feature of ECG we are interested in – are shown on the simulated V2 in Figure 3.3 and on the simulated V5 in Figure 3.4. For even lower computational complexity, the simulations on the coarse model are set to a low temporal resolution, with steps of 10 ms in the time interval from 200 ms to 500 ms after the excitation start – a total of only 31

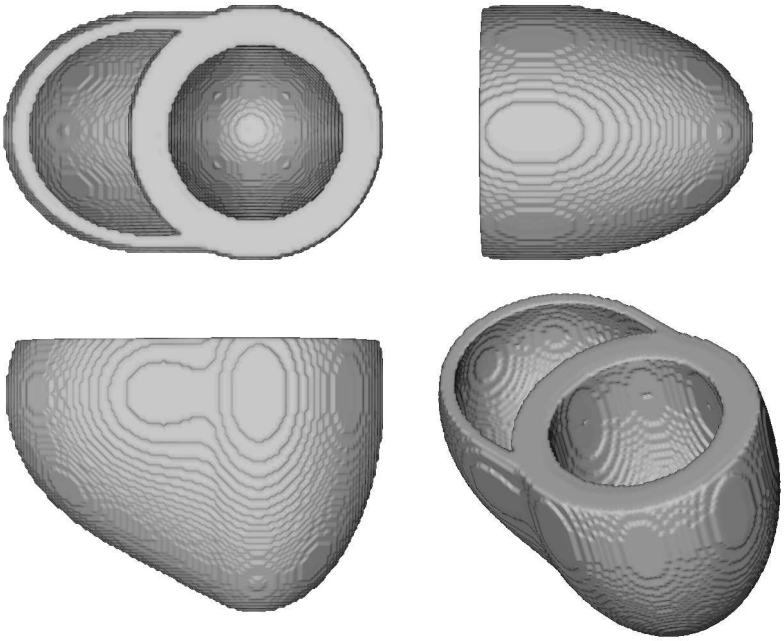

Figure 3.2: Improved heart model, featuring a larger left ventricle (the main thick oval), and a smaller right ventricle (the thinner oval attached to the main one), cut just below the atria, which are not modeled.

steps.

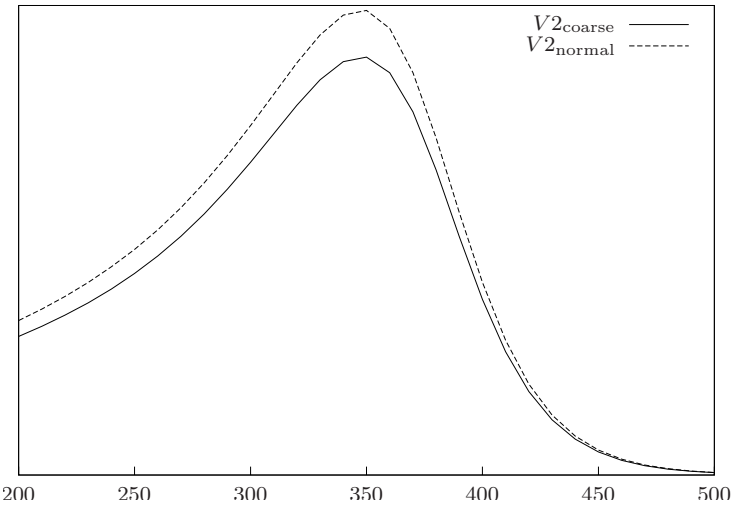

Figure 3.3: Comparison of the simulated ECG on the electrode V2 using the coarse (2 mm spatial resolution) and the basic (1 mm spatial resolution) heart models. Vertical scale is unlabeled because the realistic amplitudes of ECGs are not simulated.

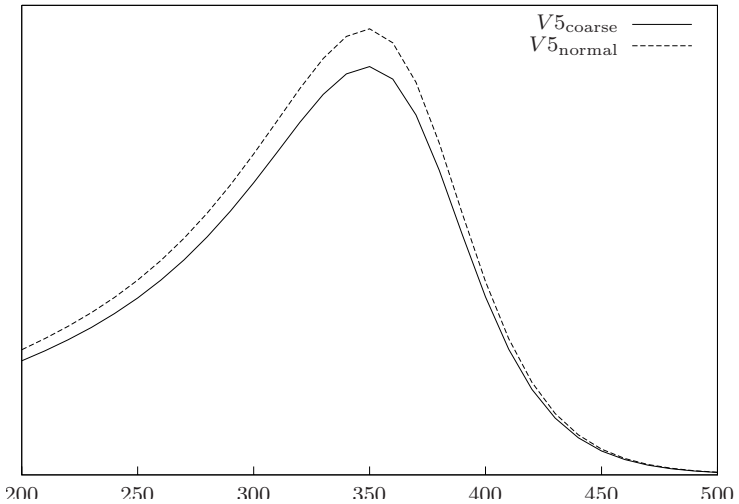

Figure 3.4: Comparison of the simulated ECG on the electrode V5 using the coarse (2 mm spatial resolution) and the basic (1 mm spatial resolution) heart models. Vertical scale is unlabeled because the ECGs are not simulated with realistic amplitudes.

For the additional speedup of the optimization, the objective function is composed of two steps. In the first step, the initial simulation is performed using a one-dimensional heart model (also called string model) composed of only two cells. If needed, In the second step an extensive simulation is performed using the above described three-dimensional heart model. The initial simulation on the simplified string model is very fast but also inaccurate and can only simulate a single ECG at a time. There is however a strong correlation between the ECG simulated using the string model and the ECGs simulated using the three-dimensional model on both selected electrodes (see Figure 3.5), making the string model suitable for filtering out extremely bad solutions.

The objective function first calculates the Pearson correlation coefficient between the ECG simulated using the string model and the ECG measured on V2. If the coefficient is negative (indicating a very poor solution), the objective function terminates, returning both objectives equal to this coefficient. Otherwise the extensive simulation is performed using the three-dimensional model, calculating both objectives separately and accurately. The correlations between randomly generated solutions and the measured ECGs are expected to be distributed equally between positive and negative, causing the second step of the objective function to be skipped half of the time on the initial random population. Figure 3.6 shows how the ratio of solutions entering the second stage of objective function increases with the number of performed evaluations. The chance of creating very bad solutions that would not pass the first stage of evaluation evidently drops dramatically as the quality of solutions in the population increases. Still, the two phase evaluation should lower the time requirements for optimization.

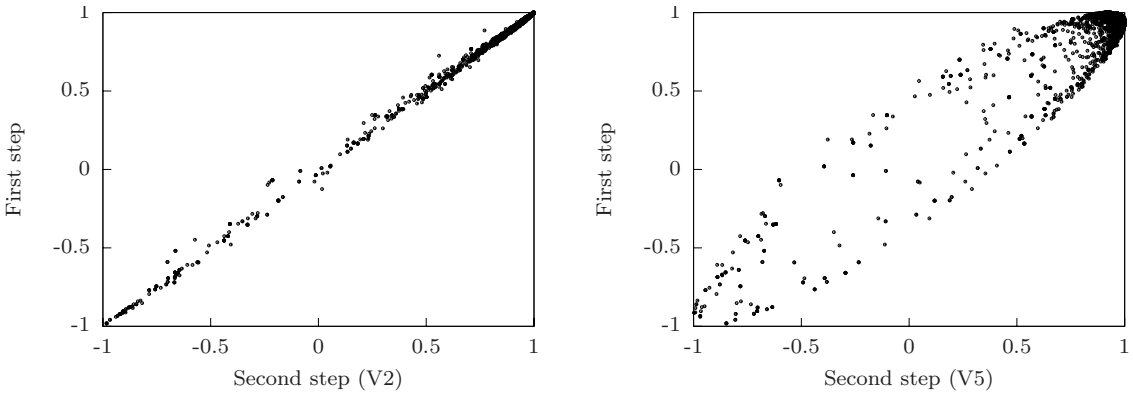

Figure 3.5: The correlation between results of the first and the second step of evaluation function shown separately for both objectives. While the second step is done for a different measuring positions for each objective using the three-dimensional model, the first step is calculated using a string model and its results apply to both objectives. 20000 randomly selected evaluations, that were performed during tests are shown. Their distribution is typical for a single optimization run on a single processor.

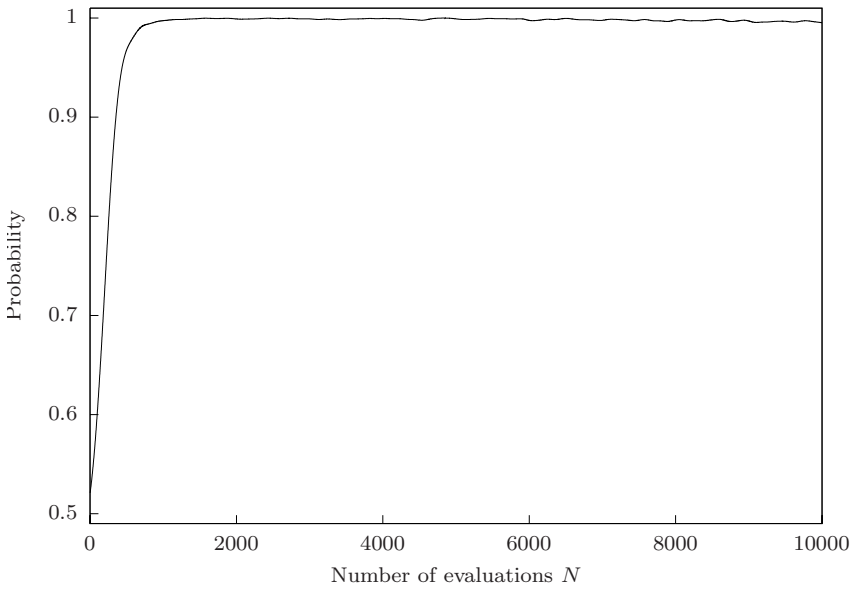

Figure 3.6: Probability for a solution generated after N evaluations to enter the second stage of the ECG problem objective function. The source for this graph are all the generated solutions from 100 runs, with 10000 evaluations each. The graph is smoothed with a Gauss filter with $\sigma = 32$ for clarity.

To get realistic positioning of observation points, the heart model is placed into a virtual elliptic cylinder that represents human torso. Observation points are then placed on this cylinder, simulating the placement of electrodes for a multichannel ECG measurement [10] as shown in Figure 3.7. The simulated ECGs obtained with the described model can be compared to the multichannel ECG measurements, allowing for more rigorous tests of the simulator than would be possible using only the standard 12-lead ECG measurements. Observation points corresponding to V2 and V5 in measurements are used in the presented optimization scenario.

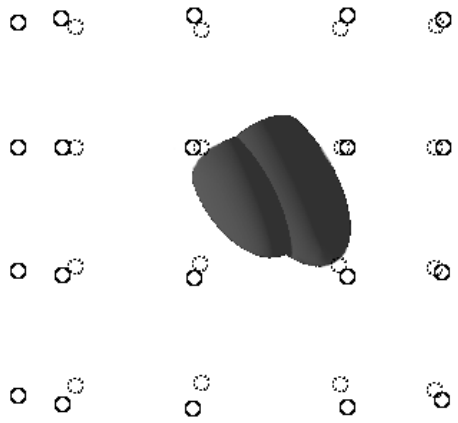

Figure 3.7: Front view on the observation points on the torso relative to the improved heart model. Points in front of the torso are marked with circles, points on the back of the torso with dashed circles. Positioning of the observation points simulates the placement of electrodes for a multichannel ECG measurement.

In the experiments for [24], the ECG problem required setting three unknown APs, for three AP layers, with three free parameters each, in a way that maximized the fidelity of simulated ECGs. We further propose a modification of AP model [63, 64]. We replace one of the sigmoidal function with an asymmetric sigmoidal function, which requires an additional parameter to control the asymmetry. Since then, we have also changed the simulator to only require setting two unknown APs, thus the optimization problem described in this chapter has eight parameters and the candidate solutions are encoded as 8-dimensional real-valued vectors.

Chapter 4

Parallel Algorithms Based on DEMO

In this chapter, two parallel implementations of DEMO are presented. We use the variant DEMO/parent described in [55], using DE/rand/1/bin scheme as the base algorithm because it already proved comparable to the state-of-the art algorithms for multiobjective optimization [54]. Used on the continuous casting optimization problem [30], however, proved very time consuming, with single run taking more than 3 days to complete on an average PC. To speed it up, it was parallelized to run on a computer cluster, using message passing as a means of communication. Given the properties of the available cluster of 16 identical dual-processor computers with fast interconnections, master-slave parallelization model was selected as the most appropriate and has been used in both parallel algorithm implementations. First, *generational DEMO* algorithm is presented, which uses a modified DEMO and the standard master-slave parallelization. It is used as a baseline for comparison with the more sophisticated *AMS-DEMO* algorithm, which uses the original DEMO and a modified master-slave parallelization type.

4.1 Generational DEMO

First we present a parallel algorithm for numerical multiobjective optimization on homogeneous parallel computer architectures. It is based on DEMO that is first converted from steady-state to generational and then parallelized using the standard master-slave parallelization type. Although designed for use on homogeneous parallel computer architectures, it can use heterogeneous architectures as well, but with lower utilization of faster processors. This algorithm has also been published in [29].

4.1.1 Description

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 gener-

ations, where in each iteration fitness values are calculated for individuals of the current population and the evolutionary algorithm operators are applied to them, creating 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 is 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, requiring interprocess communication, which is implemented as message passing, 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. Coupled with multiple communication couples per generation, this causes some unnecessary synchronizations. After the fitness values for all individuals are known, the master process applies the evolutionary algorithm operators and creates 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 generational DEMO is, in the context of multiobjective optimization, known as the Parallel Function Evaluation (PFE) variant of the single-walk parallelization [45]. It is aimed at speeding up the computations, while the basic behavior of the underlying algorithm remains unchanged.

4.1.2 Estimated Execution Time

What will be the expected benefits of generational DEMO running on several processors in comparison to generational DEMO or the original DEMO 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. We provide an analytical model for prediction of execution times of generational DEMO in dependence of the number of processors and the evaluation time. In the model, we assume evaluation to be the most demanding part of DEMO, making the time it takes to execute other parts of the algorithm negligible in comparison. Two conditions have to be fulfilled for this assumption to hold.

The first one is that the evaluation time has to be significantly longer than the communication time. On modern hardware, communication time is in the order of milliseconds or less, making this evaluation dependent and thus problem related. The condition holds for most real-life problems, including our test problems. On the other hand, it does not

Algorithm 4.1: Generational DEMO – master process

```
1  create an empty initial population ℙ
2  while stopping criterion not met do
3      create an empty population ℙ_new
4      if ℙ empty then
5          fill ℙ_new with popSize random solutions
6      else
7          foreach solution P from ℙ do
8              randomly select three different solutions I_1, I_2, I_3 from ℙ
9              create a candidate solution C := I_1+F·(I_2 − I_3)
10             alter C by crossover with P
11             Parent (C) ← P
12             add C into ℙ_new
13     repeat
14         n ← min(number of unevaluated solutions in ℙ_new, number of slaves + 1)
15         select n unevaluated solutions from ℙ_new: C_1..C_n
16         send C_2..C_n to slaves into evaluation
17         evaluate C_1
18         receive fitnesses of C_2..C_n from slaves
19         for solutions C_i, i = 1..n and their parents from ℙ_new do
20             if C_i dominates Parent (C_i) then
21                 leave C_i in ℙ_new
22             else if Parent (C_i) dominates C_i then
23                 replace C_i with Parent (C_i) in ℙ_new
24             else
25                 add Parent (C_i) into ℙ_new
26     until all solutions from ℙ_new evaluated
27     if ℙ_new contains more than popSize solutions then
28         truncate ℙ_new
29     ℙ ← ℙ_new
30 send termination request to all slaves
```

hold for test problems, such as those introduced in [71], which are defined as evaluation functions which can be calculated extremely fast.

The second one is related to parallel setup. Although parallel DEMO variants require very little computation other than solution evaluation, there is a computationally

Algorithm 4.2: Generational DEMO – slave process

```
1 while termination not requested do
2     wait to receive a message from master
3     if the message contains an individual C for evaluation then
4         evaluate C
5         send the fitness of C to the master
```

expensive step that can require significant execution time. This is the calculation of the crowding distance metric, which is required for population truncation, and has computational complexity of $O(Mn \log n)$, where M is the number of objectives and n is the population size. It could amount to significant execution time on very large n and/or M.

Assuming the evaluation to be the most demanding part, the execution time of generational DEMO t_{gen} equals the number of generations N/n times a 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 $t_{par}(p)$, repeated $\lfloor n/p \rfloor$ times, plus a parallel evaluation of the remaining $n \bmod p$ solutions on p processors:

$$t_{gen} = \frac{N}{n} \left(t_{par}(p) \left\lfloor \frac{n}{p} \right\rfloor + t_{par}(n \bmod p) \right) . \tag{4.1}$$

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

$$t_{par}(m) = E\left(\max_{i=1}^{m} \{t_{e,i}\} \right) . \tag{4.2}$$

It can be approximated as the mean of the cumulative distribution function (CDF) of maximum time of m evaluations, which equals the CDF of the solution evaluation time, raised to the power of m. An important note to the parallel evaluation time is, that it does not only increase with the mean evaluation time but also with the evaluation time variability, causing a drop in performance if either the optimization problem or the parallel architecture causes the evaluation time to vary. The effect can be observed if the Equation 4.1 is plotted against p for varying evaluation time variability, as the example in Figure 4.1 clearly shows.

On one hand, our estimated execution times indicate a very good speedup of generational DEMO on homogeneous computer architectures and large numbers of processors. The algorithm is limited, on the other hand, by the requirement that p divides population size, and does not handle variations in evaluation time well. The next section describes a new parallel algorithm, which is devised primarily to overcome these two limitations.

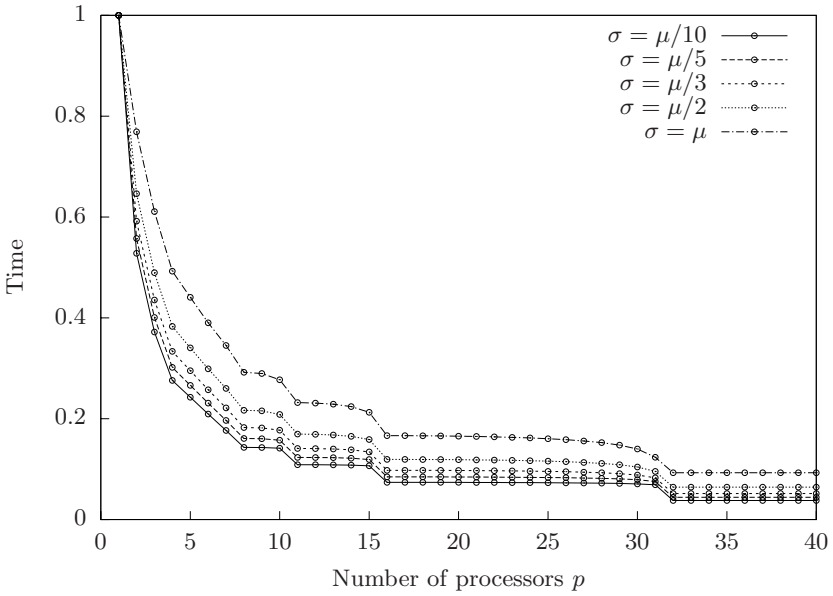

Figure 4.1: Generational DEMO execution time vs. the number of processors p, with varying standard deviation of evaluation time. Time is shown relative to the execution time on a single processor. In the legend, σ stands for standard deviation of t_e and μ stands for mean value of t_e. The results indicate a steady degradation in performance with increasing σ.

4.2 AMS-DEMO

By shifting the main task of the algorithm from traversing the search space in the same way as it would on a single processor, towards keeping the slaves constantly occupied, we create a new parallel algorithm, called *AMS-DEMO*, which greatly exceeds the flexibility of generational DEMO. AMS-DEMO however remains effective only on problems with long evaluation time.

4.2.1 Description

Conceptually, AMS-DEMO works as p asynchronous and independent DEMO algorithms working on a shared population, as shown on the right hand side of Figure 4.2. The master-slave model is used for parallelization, with the slaves running on all of the p processors and the master running as an additional process on one of the processors. Population is stored on the master, where the variation operators and selection are also applied, while the slaves only evaluate the solutions supplied to them by the master as shown on the left hand side of Figure 4.2. Because the slaves operate asynchronously, there is no need for them to be load-balanced. In practice this means AMS-DEMO is able to use heterogeneous computer systems, computers with varying background load, dynamic number of processors, and that there are no performance based restrictions on the population size and on the number of processors. Another advantage of the

asynchronous nature of the algorithm is its robustness to a loss of a processor during the execution, for example, due to network related problems.

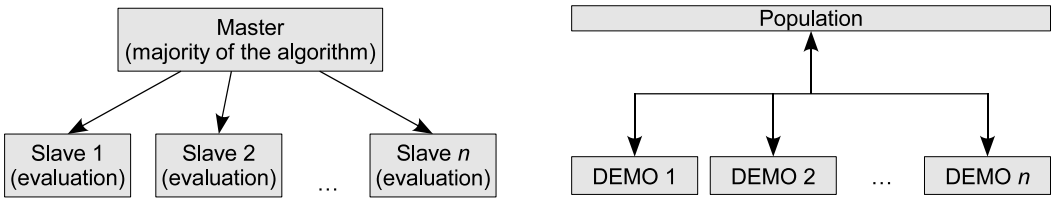

Figure 4.2: Standard division of operations between the master and slaves for the master-slave parallelization model (left), which holds for both generational DEMO and AMS-DEMO. In contrast to the master-slave algorithms being equivalent regardless of the number of processors they run on, AMS-DEMO exhibits a different behavior (right). It is equivalent to several serial DEMO algorithms, one for each processor, sharing a population while being independent in other respects.

All the slaves only wait a minimum amount of time between the evaluations, while the master, on the other hand, performing operations orders of magnitude shorter, spends most of the time waiting. This, however, does not decrease the efficiency of the algorithm, since the master shares a processor with one of the slaves, causing that processor to never be idle, by either executing the master or the slave process. When running on a single processor, even though separated into two processes, the algorithm behaves identically to the original DEMO algorithm.

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 manifests as the ability of the sender and 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, which synchronizes them. This requires the sender to wait for the receiver to start listening before it can pass on the message, and the receiver to anticipate the message. Generational DEMO requires the processors to be synchronized at the time messages are either gathered from the slaves or sent to the slaves, making the synchronous message perfect for the task. For AMS-DEMO, the asynchronous message passing performs better, because it spends no time waiting on synchronization.

To utilize the asynchronous communication to full extent, AMS-DEMO introduces local queues of solutions pending evaluation to 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 the 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 of any pending messages from the master. Both of these operations are fast because of the asynchronous message passing.

Note that in our implementation, 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 – the slave only removes it after it has evaluated it. 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 possible exceptions to this rule, 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.

4.2.2 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 be easily demonstrated if we observe a solution from its creation to its selection. In the original DEMO, the population does not change in this observed time period. In AMS-DEMO, while the observed solution is being evaluated on one processor, some number of other solutions may complete evaluation on other processors. If they survive the selection, these solutions change the population in between the observed solution creation and its selection. This causes a lag in exploitation of good solutions. We will call it *selection lag* and define it per solution as the number of solutions that undergo selection in the time between the observed solution's creation and selection. Selection lag $l(s)$ 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 s, but would be known to the original DEMO under the same circumstances. Because every selection is coupled with the creation of a new solution, selection lag can also equals the number of solutions created while an observed solution is being evaluated. In other words, selection lag of a solution is the number of solutions that could be created differently in the original DEMO than they are in AMS-DEMO, because the processing of the observed solution differs between these two algorithms.

It should be stressed that the changes to the population counted by selection lag are possible, but not necessary. Furthermore, although the probabilities of changes depend linearly on selection lag, they also depend on the probability for the offspring to survive selection. Finally, a change in the population does not always cause a deviation of the AMS-DEMO search path relative to the original DEMO search path. The probability of causing it depends on the size of the population. For larger populations, for example, variation operators have increased probability of selecting solutions that remained

unchanged during the last $l(s)$ selections, therefore making any change to the population irrelevant.

Defined per solution, selection lag does not say much about the algorithm as a whole, only about one of the solutions processed by the algorithm. Yet, individual selection lags accumulate to significant changes in AMS-DEMO behavior compared to the original DEMO behavior. We therefore define selection lag of AMS-DEMO as the value of function $l(s)$ across all solutions processed in a single run. Its most important property is its mean, which equals $pq-1$, where p is the number of processors (which in turn equals the number of slaves) and q is the queue size (equal on all slaves). We show this equality through an example. In the simplest case, with queue size 1 and equal evaluation times, the slaves work as follows. They are assigned the solutions and start evaluating them in an orderly fashion. Slave 1 is assigned solution 1, then slave 2 is assigned solution 2, and so on until the last slave p is assigned solution p. Because all evaluation times are equal, the slaves finish evaluations and receive the next set of p solutions in the same order as they received the first set. Slave 1 is assigned solution $p+1$, slave 2 solution $p+2$ and so on. Therefore, selection lag for all solutions, given as the number of solutions created during their evaluation, equals $p-1$. Mean selection lag is then also $p-1$.

If the evaluation times are allowed to vary in the given example, selection lag may no longer equal $p-1$ for all solutions. Mean selection lag, however, remains $p-1$ due to the fact that any increase in one solution's selection lag must produce an equivalent decrease in selection lags of other solutions. This is also best shown on an example of two solutions, a and b, evaluated in parallel, with a undergoing selection just prior to b. a has selection lag l_a and b has selection lag l_b. If the evaluation time of a were to increase just enough for it to undergo selection after b instead of prior to b, l_a would increase by 1. But this change in evaluation time of a would also influence b. Its selection lag would necessarily decrease by 1, because one solution less, namely a, would undergo selection, while b was being evaluated. Thus any transposition of the evaluation order of two solutions changes their selection lags symmetrically, keeping their mean selection lag unchanged. 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.

On the introduction of queues, the time between creation and selection of a solution lengthens by the time the solution in question waits in queue. The number of solutions generated in this time equals the number of solutions in front of the queue $(q-1)$, plus the number of solutions generated in this time on other processors $(q-1)(p-1)$. The average solution selection lag or the total number of solutions generated in the time the solution in question is waiting or being evaluated is therefore $(q-1)+(q-1)(p-1)+(p-1)$, which can be simplified to $pq-1$.

For the final word on selection lag, it should be noted that although selection lag fully explains the changes of the original DEMO algorithm arising from the parallelization,

it does so only if its full distribution is observed. Its mean is only the most important characteristic, with the benefit of being easily calculated. We can assume, however, that as long as selection lags of solutions deviate little from the mean, which they do on our test problems, the errors we make by observing only mean selection lags are negligible.

4.2.3 Implementation Details

Although the master process of AMS-DEMO mainly implements the functionality of the original DEMO algorithm, the inclusion of communication and slave supervision complicates it somewhat. Its pseudocode is listed in Algorithm 4.3, Function 4.4 and Function 4.5.

Algorithm 4.3: AMS-DEMO algorithm – master process

```
1  create empty queues Q_1 ... Q_p   /* local copies of the queues on slaves */
2  create empty population ℙ
3  parentIndex ← 1
4  numEvaluated ← 0
5  while stopping criterion not met do
6      while ∃k : |Q_k| ≤ minimumQueueLength do
7          c = Create (parentIndex)
8          select index j such that ∀k ∈ [1 ... n] : |Q_j| ≤ |Q_k|
9          add c to Q_j
10         send a message containing c to slave S_j
11         parentIndex ← (parentIndex + 1) mod popSize
12     while no pending messages from slaves do wait
13     extract solution c from the first pending message
14     remove c from Q_j, where j is the number of the slave that sent the message
15     Selection (c)
16     numEvaluated ← numEvaluated + 1
17     if numEvaluated is a multiple of popSize then
18         truncate ℙ to contain no more than popSize solutions
19         randomly enumerate solutions from ℙ
20 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 and thus avoids the required synchronization of processes associated with such a step. AMS-DEMO rather starts with

Function 4.4: Create(i)

Input: **global** population $\mathbb{P}$ of size *popSize*
global algorithm parameter F
index of the parent i

Result: creates a new candidate solution **c** and stores a link to its parent

```
1  if |ℙ| < popSize then
2      randomly create a solution c
3      mark c as unevaluated
4      mark c as the parent of itself
5      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 return c
```

Function 4.5: Selection(**c**)

Input: **global** population $\mathbb{P}$
candidate solution **c**

Result: modifies $\mathbb{P}$ by performing selection between **c** and its parent

```
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  if p is not evaluated then
5      replace p with c
6  else
7      compare c to p
8      if c dominates p then
9          replace p with c
10     else if p dominates c then
11         keep p
12     else
13         keep p and add c to ℙ
```

an empty population $\mathbb{P}$ and modifies the way it creates solutions in its main loop, as it can be seen from Function 4.4, to allow for different creation of solutions of the initial population from those of the subsequent ones.

Because the solutions of the initial population are now created in the main loop, but they do not have parents to compete against, the process of selection must be able to detect them and allow them to bypass it (see Function 4.5). Before the offspring **c** and its parent **p** enter the selection, the parent is examined (line 1 of Function 4.5). If it is marked as unevaluated, then the offspring bypasses the selection and directly replaces the parent. There are two possible scenarios resulting in the parent being marked as unevaluated. The first and the more common one is that the offspring is a part of the initial population, having no parent, and was marked as the parent of itself when it was created (line 3 of Function 4.4). The second and rarer scenario is, that the parent is a member of the initial population and has not finished evaluating yet. Possibly because it is being evaluated on a one of the slower processors comprising the parallel computer. 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.

The slave process of AMS-DEMO is much simpler as it only does evaluations of solutions supplied by the master. Its pseudocode is listed in Algorithm 4.6. It consists of two main parts – checking for and processing of messages from the master, and evaluating solutions. Messages from the master can be of two types. Messages of the first type contain a solution to be evaluated, while the messages of the second type contain a request for termination. The slave receiving the latter terminates immediately, even if there are solutions waiting in its queue.

Algorithm 4.6: AMS-DEMO algorithm – slave process

```
1  create empty queue Q
2  while termination not requested do
3      if Q empty then
4          while no pending messages from master do
5              wait for a message
6          push solutions from received messages into Q
7      else
8          evaluate the top element of Q
9          send the evaluation results to the master
10         pop the top element of Q
```

4.2.4 Estimated Execution Time

The AMS-DEMO execution time (wall clock time) is calculated from the sum of execution times of individual processors, divided by the number of processors p. Because we work under the assumption of evaluation time being orders of magnitude longer than times of communication, output operations, and DEMO operators, all of which can therefore be safely neglected, we can say the sum of execution times of all processors equals the sum of all evaluation times (which equals the average evaluation time $\overline{t_e}$ times the number of evaluations N) plus the sum of all idle times. Since the master does not perform evaluations and shares a processor with a slave, we can ignore it in this calculation. The slaves are only idle at the end of the optimization, from the time the first slave has finished its last evaluation, until the time the last slave has finished its last evaluation. Assuming equally fast processors and slightly varying t_e, the slaves that started evaluations at the same time will end up performing evaluations completely asynchronously – at any moment in time, each one will have a different portion of its current evaluation already performed. Given enough evaluations have passed, the portions of current evaluation already performed will be distributed uniformly on all the slaves. When the solution that fulfills the stopping criterion is evaluated and received by the master, the slave that evaluated it has 0 idle time. The other $(p-1)$ slaves are terminated at this point, causing all the work on their current evaluations to be discarded and the time they spent performing their last evaluations can be counted as idle time, since it produced no useful results. They experience idle time that on average equals the mean of proportion of evaluation already performed times the mean evaluation time: $0.5\overline{t_e}$, making total idle time of the system $t^{\text{idle}}_{\text{AMS}} = 0.5(p-1)\overline{t_e}$. Although we analyzed the scenario in which the stopping criterion is based on the solution quality, similar holds if the stopping criterion is a predefined limit on the total number of evaluations. The only difference is that in the latter case, the other $(p-1)$ slaves no longer perform evaluations that are interrupted and then discarded, but rather do not receive any new solutions to evaluate and are actually idle. Putting the partial equations together and then simplifying, the estimated execution time for AMS-DEMO is:

$$t_{\text{AMS}} = \frac{\overline{t_e}N + 0.5(p-1)\overline{t_e}}{p} = \frac{\overline{t_e}(N + 0.5(p-1))}{p} . \tag{4.3}$$

Note that since the estimated working or non-idle time depends on N, the estimated idle time does not. It is constant with respect to N, causing the ratio of idle time versus non-idle time to decrease with increasing N. AMS-DEMO therefore experiences relatively less idle time and is more efficient, the more evaluations it has to perform. Total idle time can also grow quite large even when compared to wall clock time. In Figure 4.3 we plot t_{AMS} and $t^{\text{idle}}_{\text{AMS}}$ versus p for $N = 1000$. Although $t^{\text{idle}}_{\text{AMS}}$ is the sum of idle times on all processors and as such not directly comparable to t_{AMS}, the two graphs plotted together

give insight into the expected efficiency of AMS-DEMO. Significant amount of processor time can be spent waiting even though AMS-DEMO synchronizes processors only once – at the end of the execution. At $p = 45$, $t^{\text{idle}}_{\text{AMS}}$ becomes greater than t_{AMS}, the described parallel system performs as if 44 out of the total 45 processors was constantly busy and 1 was constantly idle.

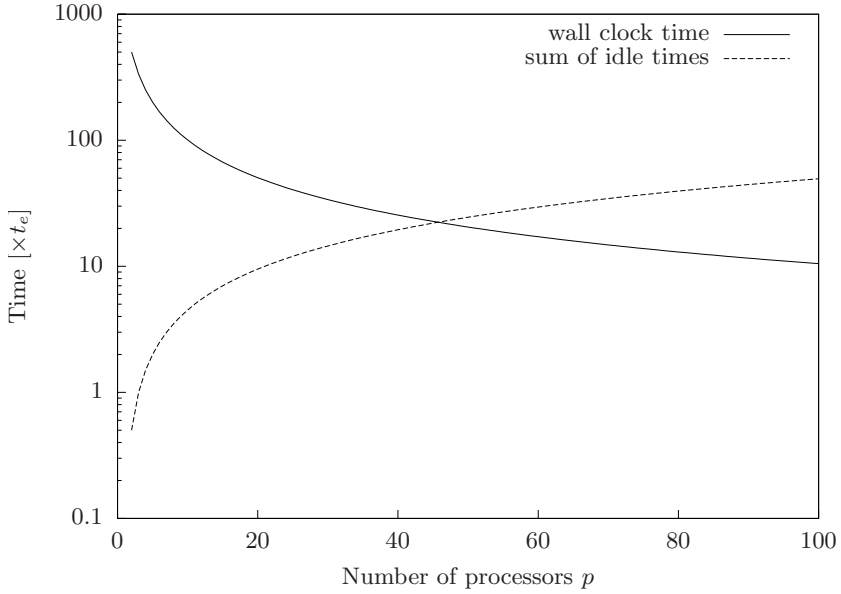

Figure 4.3: AMS-DEMO wall clock time as the function of the number of processors p when the total number of evaluations N is set to 1000. Although they are not directly comparable, the sum of idle times for AMS-DEMO is also shown on the same axis. Both graphs are plotted relative to evaluation time t_e.

To end the chapter on parallel algorithms, a short summary is given. To speedup DEMO, two parallel variants were implemented, both based on master-slave parallelization type. Both were described in detail in this chapter. First variant – generational DEMO – extends a modified (from steady-state to generational) DEMO using the standard master-slave parallelization type. Its main properties include an easy to implement parallelization and deterministic behavior independent of the number of processors. It executes efficiently only when the population size is a multiple of the number of processors, the processor set is homogeneous and the evaluation time is constant. Generational DEMO is used as a baseline for comparison with the more sophisticated second parallel variant of DEMO – AMS-DEMO, which extends an unmodified DEMO using a modified master-slave parallelization type. AMS-DEMO executes efficiently on sets of homogeneous as well as heterogeneous parallel computers, on any number of computers, on dynamically changing number of computers and when evaluation time is variable. In

contrast to generational DEMO, AMS-DEMO also executes identically to DEMO on a single processor. Its path through search space then increasingly deviates with every additional processor, which is expected to slow down the convergence rate. The measure of this deviation is named selection lag and a way of calculating its expected value is given. The equations for estimating execution time of both algorithms are also given and will be used to support analysis in the next chapter.

Chapter 5

Numerical Experiments and Results

This chapter first presents the experimental setup and the tests carried out to empirically evaluate AMS-DEMO and to compare it to generational DEMO. Next, the results are presented, starting with the results of the optimization, followed by the analysis of AMS-DEMO convergence and parallel speedup, both measured experimentally. Following are the analytical projections of the behavior of both algorithms on untested number of processors. At the end, a test on a heterogeneous computer architecture is presented.

5.1 Experimental Setup

The implementation of the presented algorithms is in C++ and compiled with *gcc v3.3.3* for target 64-bit *Linux* system. For interprocess communication, MPICH library [35] version 1.2.7 is used, which is the implementation of MPI (Message Passing Interface) standard [58]. A cluster of 17 dual-processor nodes (each node being a personal computer) is used for empirical evaluation. Each node contains two AMD Opteron 244 processors, 1024 MB of RAM, a hard disk drive, six 1000 MB/s Full Duplex Ethernet ports and an independent installation of the Fedora Core 2 operating system. During the experiments, all nodes are required to be running only the background system processes which leaves nearly all capabilities to be used by the algorithm. The nodes are all interconnected through an Ethernet switch, and, in addition, there are several direct interconnections between the nodes (see Figure 5.1). Nodes 1 through 16 are connected by a toroidal 4-mesh, and nodes 1 through 4 are directly connected to node 17. This node also serves as a host node, through which users access the cluster. Switch is used to connect pairs of nodes that are not connected directly. This makes the use of any desired topology possible. In our tests, star topologies of various sizes were used.

When selecting multiple computers from the cluster for experiments, our choice of computers was based only on their availability and we did not try to use sets of computers with physical interconnections that form a star topology. Although this fact could

cause some difference in run times between runs on different subsets of computers, it is negligible, because, as we will show later on, the total communication time is orders of magnitude shorter than the total run time and is actually within the run time variability. Tests using a single processor were performed in pairs on a single computer when possible, each processor performing its own test run. We also never used only one of the processors of a computer for a test run that required multiple processors. Since all of our tests on multiple processors were executed on an even number of processors, we always used both processors on all the used computers. All of the performed tests required multiple runs, which were usually performed on several different sets of computers – again, selected depending on their availability. An exception were tests on 32 processors which were always performed on the same set of computers. We noticed no significant differences between runs of tests performed on different set of computers.

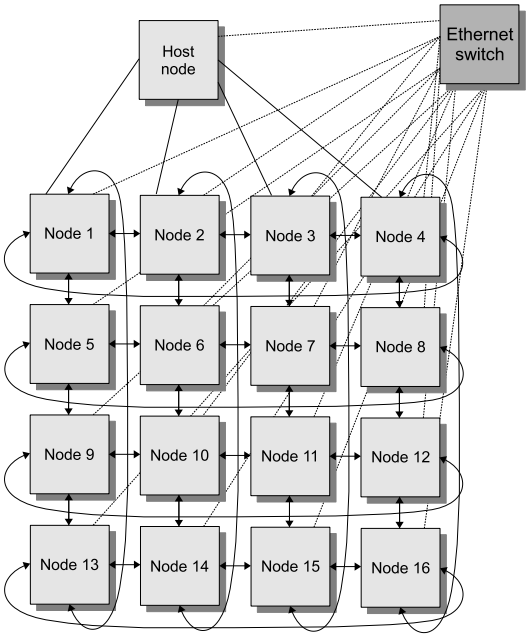

Figure 5.1: Architecture of the computer cluster used in tests. All nodes are interconnected either directly or through the Ethernet switch, forming a fully connected graph.

5.1.1 Varying the Queue Length

Although the queues have been implemented in AMS-DEMO to reduce the slave idle time to a minimum, they also allow simulating more processors than are available on the test architecture, as we will show further on. This allows for a simulation of a curious AMS-DEMO 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, queues are chosen to simultaneously expose the drawbacks of their use.

AMS-DEMO running on p slaves, each having a queue of length q, explores the objective space in a similar fashion as if it were running on p times q slaves, each having a queue of length 1. This is because AMS-DEMO behavior changes with its selection lag, which was shown to equal $pq - 1$. The same selection lag may be obtained through different values of p and q, therefore, increasing queue length emulates the use of additional processors. Although the settings of AMS-DEMO that produce the same selection lag produce very similar behavior, there is a difference between executing the algorithm on less processors with longer queues and running it on more processors with shorter queues. If a number of solutions are inserted into a single queue, they undergo selection in the same order as they have been inserted in. On the other hand, if the same number of solutions are distributed among different processors and the evaluation time varies, they are likely to undergo selection in a different order. This manifests as an increase in the variance of the selection lag, and, although difficult to quantify, has some influence on the algorithm behavior. We believe the standard deviation to be too small for its influence to be discernible from the noise and therefore we ignore it in further analysis. We perform the analysis of selection lag in Subsection 5.3.1, where we defend the decision to ignore the selection lag variance in our experiments.

To demonstrate AMS-DEMO capability to run on a number of processors greater than population size, queues were used to emulate 320 and 640 processors on the cooling problem, and 64, 128, 256, and 512 processors on the ECG problem. All the tests with increased queue lengths were performed using the same algorithm parameters as for the basic tests with queue length of 1. The results of the tests were used directly for the analysis of convergence, while for the analysis of speedup they were combined with a prediction of run times. The queue length of 1 was used for basic tests because it produced negligible wait times for the slaves.

5.2 Optimization Results

In this section we discuss the effectiveness of the optimization – the quality of results, starting with the cooling problem, followed by the ECG problem.

5.2.1 Experiments with Steady-State Steel Casting Simulator

The optimization of the steady-state steel casting process served primarily as a test bed for a comparison between generational DEMO and AMS-DEMO and is the continuation of the work done in [25]. Therefore, the population size was first selected as the one

that suits the problem, while also being a multiple of the number of processors, allowing a good performance of the standard master-slave parallelization type based generational DEMO. As shown in previous work [26], solving the continuous casting with optimization problem 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). For tests with a generational algorithm, 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 for generational DEMO. Further algorithm settings were adopted from previous experiments [31] and were as follows: scaling factor F 0.5 and crossover probability 0.05.

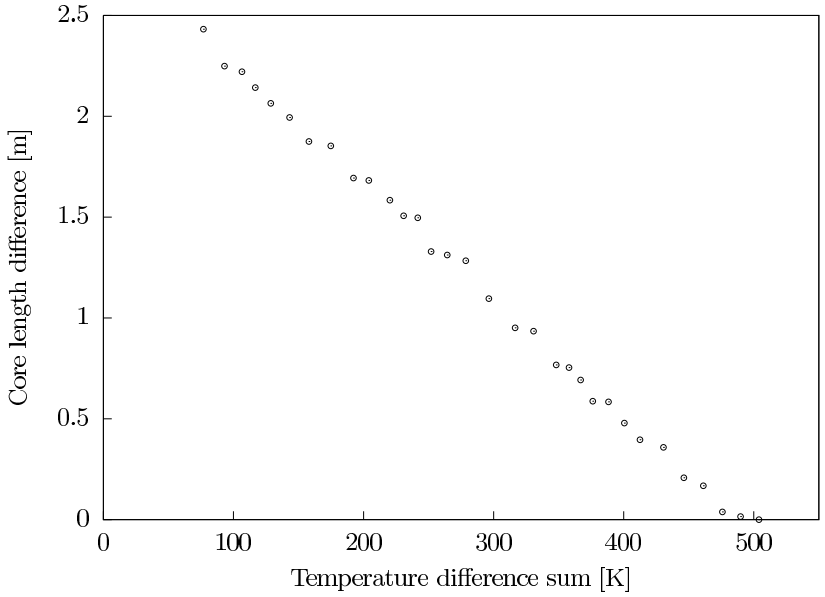

Figure 5.2: Nondominated front of solutions for the cooling problem obtained with generational DEMO.

It turned out that both parallel algorithms were able to discover the solutions known from previous applications of the original DEMO [31] demonstrating conformance with it. To illustrate the results, Figure 5.2 shows the resulting nondominated front of solutions (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 of solutions. In addition, a systematic analysis of the solutions confirms that the actual slab surface temperatures are in most cases higher than the target tem-

peratures, while the core length is shorter than or equal to the target core length. Exact temperature differences for three solutions from the front displayed in Figure 5.2, two on the boundaries of the front and one trade-off, from the center of the front, are shown in Figure 5.3. The corresponding three sets of coolant settings (parameters of the solutions) are shown in Figure 5.4.

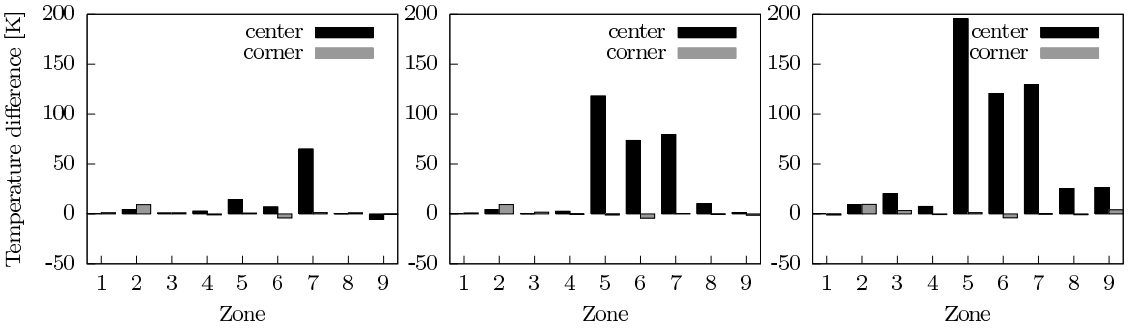

Figure 5.3: Differences from optimal temperatures in three solutions in three solutions to the cooling problem, taken from both boundaries and the center of the nondominated front of solutions, shown in Figure 5.2, sorted by their temperature difference sum. The corresponding core length deviations are −2.4 m, −1.2 m, and 0.0 m.

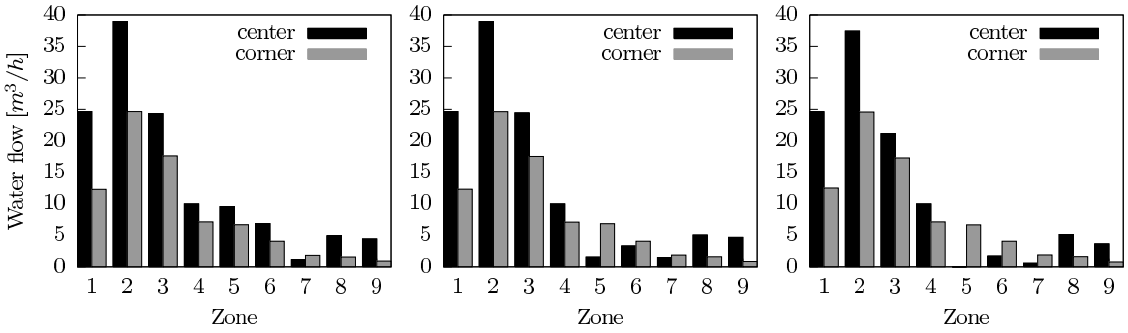

Figure 5.4: Optimized coolant flows for the three solutions shown in Figure 5.3

5.2.2 Experiments with ECG Simulator

For a thorough test of AMS-DEMO, the way its convergence rate changes with the increasing number of processors has been analyzed. A test of convergence required large number of repeated runs with varying number of processors but otherwise fixed problem

setup, to make it possible for statistical analysis to discern small changes in convergence rates. The presented ECG problem has the desired properties that enable a large number of repeated runs in a limited time frame and was therefore used mainly for the estimation of convergence rate. Generational DEMO was not tested on this problem as the test would require similar execution time as the test of AMS-DEMO, while its convergence rate is not of interest in this dissertation. But before we move on to convergence rate tests, let us observe the qualitative optimization results that we found for the ECG problem.

The obtained optimization results are somewhat surprising; the simulator always produces ECGs with nearly identical positions of the T wave peaks on both electrodes. In Figure 5.5, the optimization results are shown as the nondominated front of solutions after 10000 evaluations. The two criteria are marked as *V2 fit* and *V5 fit* and indicate how well do the simulated ECGs on electrodes V2 and V5 fit the measured ECGs on the same electrodes. Both are calculated as 1 minus correlation between the measured and simulated ECG on the observed electrode. Two extreme solutions – in which either the first or the second criterion is optimized the most – are shown in Figure 5.6 along with the measured ECG that was used as the target in the objective function. The presented solutions clearly indicate that while the simulator generates the shape of the T wave in great fidelity on two electrodes simultaneously, it is unable to generate a delay between the T wave peaks on two electrodes. This is further confirmed when all the nondominated solutions are examined. All nondominated solutions exhibit great fidelity in reproduced T wave shapes and the same timing of T waves on both simulated electrodes. This timing, which lies between the timings of both measured T wave peaks, determines the position

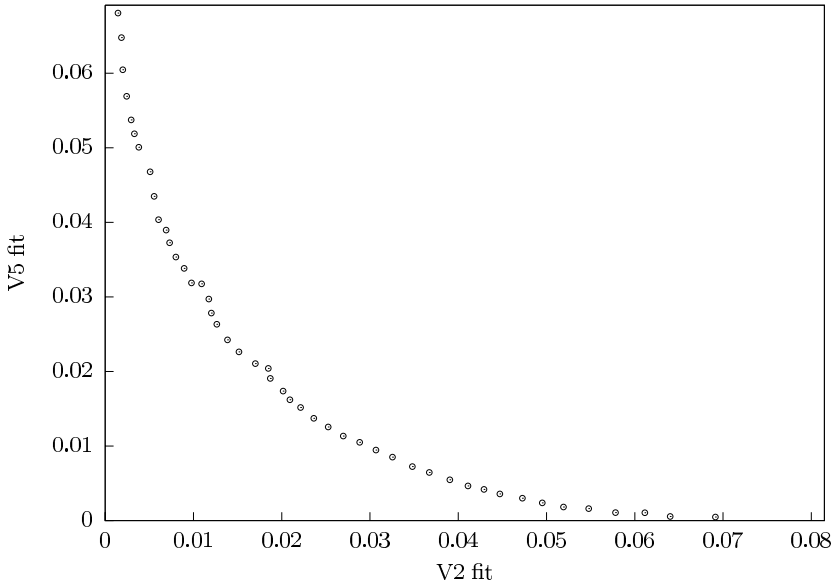

Figure 5.5: Nondominated front of solutions for the ECG problem after 10000 evaluations on a single processor.

of a solution on the front: the closer it is to the correct timing on a given electrode, the better will the objective for the same electrode be, and the worse will the other objective be.

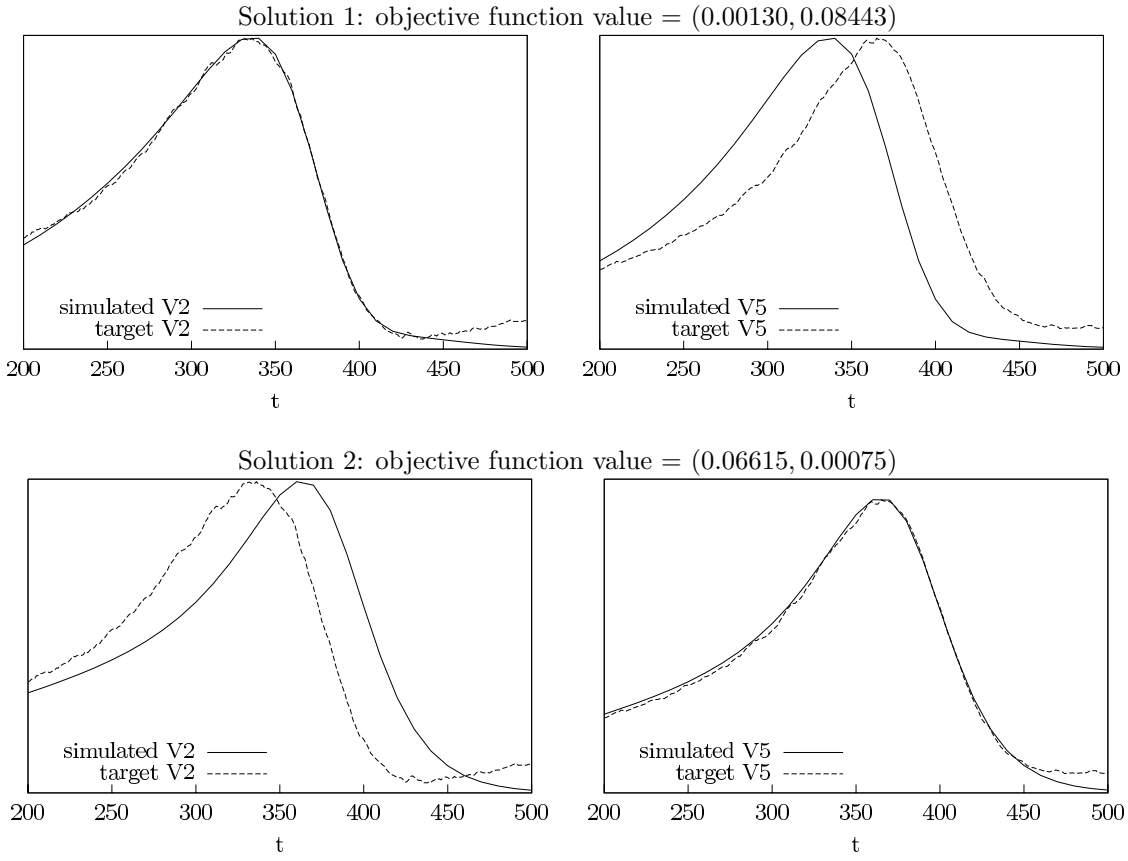

Figure 5.6: Typical solutions to the ECG problem. Because we are only interested in the shape of solutions, y axis has no unit specified and amplitudes of solutions are scaled to match the target ECGs.

How can this optimization result be interpreted? We gain two insights into the ECG simulator through it. First, the obvious one – the simulator is not able to generate accurate ECG on several electrodes simultaneously, it requires further improvements. The second insight is more subtle but also more useful; the results indicate that none of the mechanisms of ECG generation that are built into the simulator can generate the delay between T waves on different electrodes. There must be an additional mechanism responsible for the delay.

5.3 Analysis of AMS-DEMO

In this section, AMS-DEMO is compared against the serial algorithm and against the generational DEMO in a detailed analysis of the convergence and speedup on varying number of processors. To make the experimental results directly comparable, no algorithm parameters other than the number of processors varies between the tests on the same problem. Nevertheless, the tested algorithms are not equivalent even on the same parameter set, because each takes a different path through the search space, producing different results. Therefore, even though we run the algorithms for a fixed number of evaluations, we monitor their performance characteristics relative to the solution quality and not the number of evaluations.

5.3.1 Convergence

Convergence of the algorithms is characterized by the quality of the nondominated set of solutions in dependence of the number of performed evaluations. The quality of the nondominated set of solutions was evaluated using the hypervolume indicator I_H (also called the $\mathcal{S}$ metric) [73, 70], which is a measure of the hypervolume of objective space dominated by a set of solutions. A set of solutions that dominates a bigger hypervolume (a larger surface area in the case of two objectives) of the objective space is better than a set of solutions that dominates a smaller hypervolume of the objective space. The properties of the hypervolume indicator [39] enable the observation of the convergence of solutions towards the optimum within a single run, and comparison of achieved solutions between two or more runs. On the other hand, the hypervolume indicator is sensitive to the properties of the nondominated front of solutions [9], such as the uniformness of the distribution of solutions along the front, making comparison between different algorithms less reliable. The properties of the nondominated front of solutions are mainly influenced by the variation operators and the truncation of solutions, which do not differ among the algorithms we compare. The comparison is therefore valid.

In the tests of convergence, we take the original DEMO as the baseline for comparison with both parallel algorithms. For the cooling problem, we perform 25 runs of each algorithm with the stopping criterion set to 9600 evaluations, and for the parallel algorithms, we take the number of processors $p = \{1, 2, 4, 8, 16, 32, 320, 640\}$. The experiment would benefit from having more than 25 runs per test, providing more confident statistical results, but 25 is a practical limit because of the long execution times; for example, a single run of the original DEMO takes close to 80 hours to complete. For a more precise statistical analysis we use 100 runs of AMS-DEMO on the ECG problem with the stopping criterion set to 10000 evaluations. More runs are possible because of the shorter time per run; a single run of the original DEMO on this problem takes just over 15 hours. Because of shorter time per run and because more precise statistical analysis is possible, we also

use $p = \{1, 2, 4, 8, 16, 32, 64, 128, 256, 512\}$, which is different from p used on the cooling problem.

Convergence on the Cooling Problem

We first use the results obtained on the cooling problem to compare the convergence of the original and generational DEMO algorithms. Since p does not affect the convergence of generational DEMO, there is no need to analyze the convergence in dependence of p. Therefore we analyze the difference between the algorithms which hold for every p. Figure 5.7 shows the mean convergence rates of the algorithms characterized by mean I_H, obtained from 25 runs for each algorithm, as the function of the number of evaluations N. The differences, plotted in Figure 5.8, are found to be statistically insignificant on the whole tested range. The I_H values for the figures were calculated for every generation in case of generational DEMO and for every number of evaluations which is a multiple of population size – after every truncation of the population, for the original DEMO.

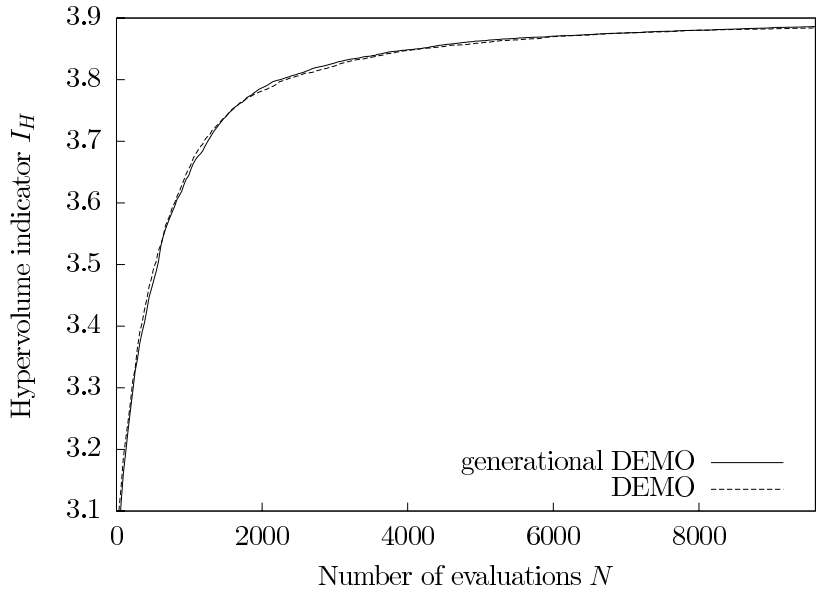

Figure 5.7: Comparison between the mean convergence rates of the original DEMO and generational DEMO on the cooling problem. Mean hypervolume indicator values I_H of 25 runs for each algorithm are plotted as a function of the number of performed evaluations N.

The results indicate that generational DEMO converges as fast as the original DEMO on the cooling problem. This means that changing DEMO from steady-state to generational did not degrade its performance and indicates a possibility for a very good performance of generational DEMO on multiple processors. Another positive effect of

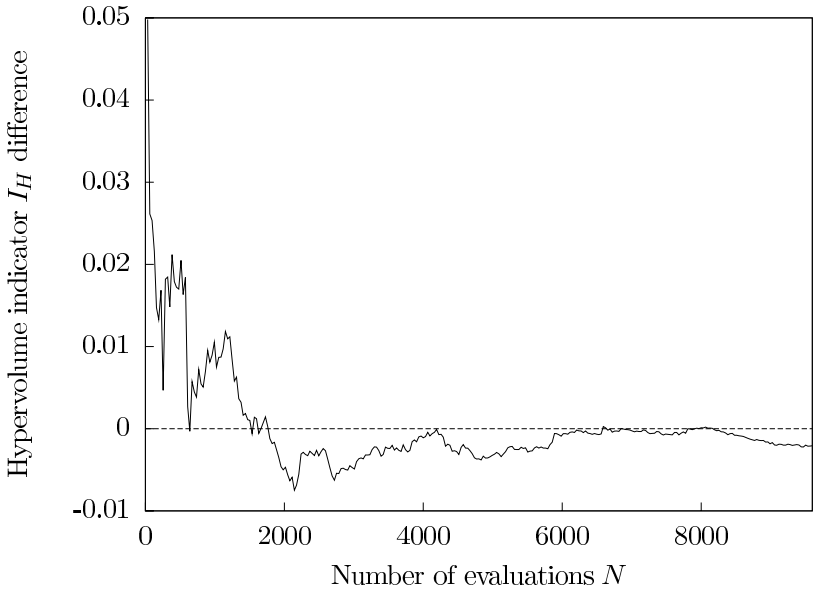

Figure 5.8: The difference between the mean convergence rates of the original DEMO and generational DEMO on the cooling problem. The value of generational DEMO hypervolume minus the original DEMO hypervolume (both means over 25 runs) is plotted as a function of the number of performed evaluations N.

the convergences not differing significantly is that in further tests, we can compare generational DEMO to the original DEMO based only on the number of performed evaluations.

Next, the convergence of AMS-DEMO is tested experimentally. It should be noted that when the number of processors drops to one, AMS-DEMO reverts back to the original DEMO – given the same random generator and seed, AMS-DEMO algorithm traverses the same path through the search space as the original DEMO, and has no calculation overhead. Therefore, tests of AMS-DEMO on a single processor are taken also as tests of the original DEMO. For each test run the value of I_H was measured after every population truncation.

Due to the changes in the algorithm, required by the parallelization, AMS-DEMO performance is expected to decrease when the number of processors increases. Figure 5.9 shows the I_H of the last population of each test run on the cooling problem. Aside from the scattering of the lower quality results and with the exception of $p = \{320, 640\}$, there is little visual difference in the distributions between the final I_H on different numbers of processors. The distributions also do not appear normal, which is confirmed with high confidence by the Kolmogorov–Smirnov test. Furthermore, the hypotheses that the underlying distributions of the last population are the same for one processor as for any other tested number of processors, cannot be rejected even at 90% confidence for $p = \{1, 2, 4, 8, 16, 32\}$, using the two-sample Kolmogorov–Smirnov tests. Therefore, the difference in results when p is less or equal to the population size n, after 9600 evaluations

appears to be too small to be discerned with the statistical tests performed on the sample size of 25 runs. On p that is 10 or 20 times larger than n, however, the lowering of the convergence rate can already be clearly seen.

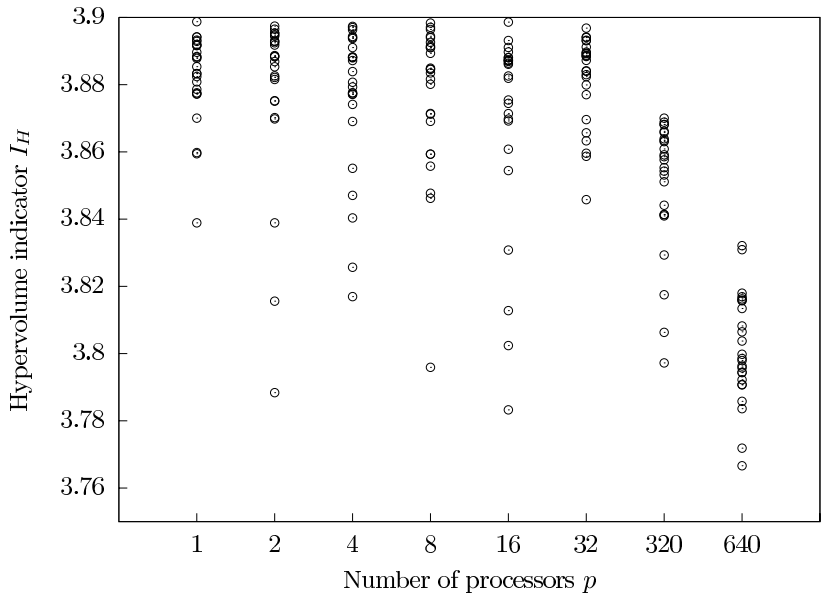

Figure 5.9: Hypervolume indicator I_H of the final populations (after 9600 evaluations) of all AMS-DEMO runs, for all tested numbers of processors p on the cooling problem.

Convergence rate is further shown in two different ways. In Figure 5.10, mean I_H is plotted against N, showing a distinctly worse convergence rate for $p = \{320, 640\}$ than for the other values of p, a group of visually similar convergence rates for $p = \{1, 2, 4, 8\}$, and slightly worse convergence rates at lower values of N for $p = \{16, 32\}$. In Figure 5.11, $I_H = \{3.5, 3.65, 3.75, 3.82\}$ is pre-specified, and the 95% confidence interval for mean N at which AMS-DEMO reaches the given I_H, are shown for all tested p. Mean values of N indicate that increasing the number of processors does slow down the convergence even for $p \leq 32$. Their confidence intervals, however, are quite large, making statistical confidence in such conclusions small. As Figure 5.9 shows, not all runs not reach $I_H = 3.82$ before they complete 9600 evaluations, when they are stopped. For $p = 640$ there are only two runs that reach it, therefore mean N is not calculated for this value of I_H. For the other 10 runs out of 175 on $p < 640$ that do not reach $I_H = 3.82$, the values of N are estimated from their value of I_H at $N = 9600$ and the average convergence rate of all the other runs for the same value of p.

The statistical significance of the differences in number of evaluations required for AMS-DEMO for specified p to reach various I_H is determined using the two-sample Kolmogorov–Smirnov test. As shown in Figure 5.12, the differences are statistically

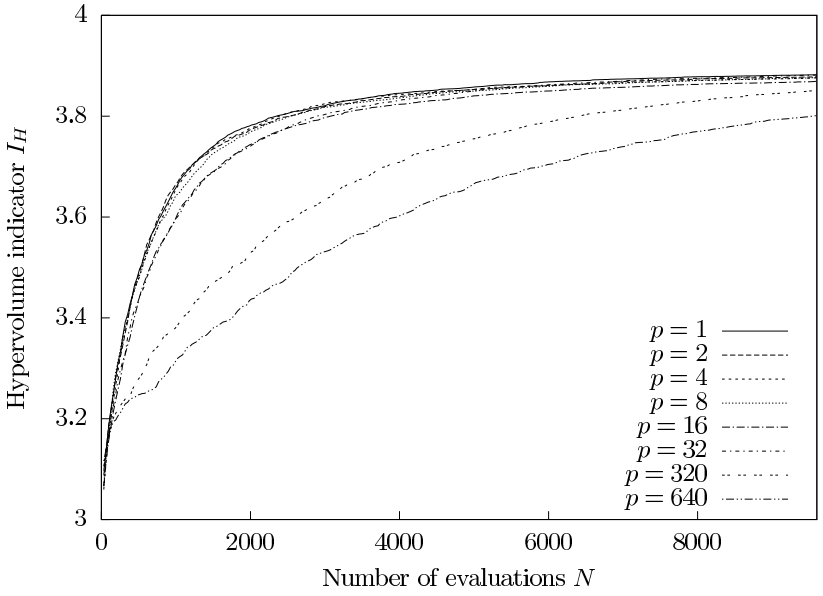

Figure 5.10: Hypervolume indicator I_H as a function of the number of evaluations N for AMS-DEMO on the cooling problem.

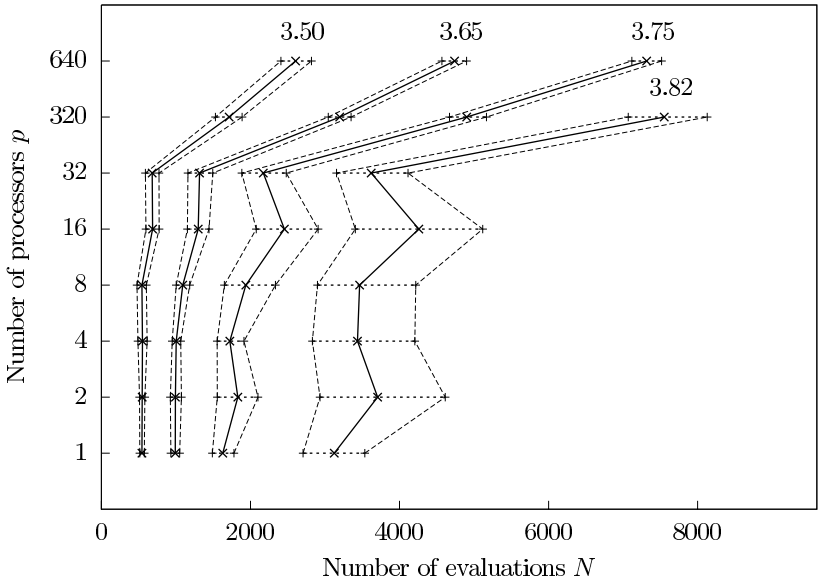

Figure 5.11: Mean number of evaluations N for AMS-DEMO to reach predefined solution quality on the cooling problem. Solution quality is specified with the hypervolume indicator I_H in labels. Best estimates for means are marked with $\times$ and connected with solid line, their 95% confidence intervals are marked with $+$, and connected with dashed lines. The value of mean N at $p = 640, I_H = 3.82$ is not available because only 2 out of 25 runs on $p = 640$ reach $I_H = 3.82$.

significant (P-value < 0.05) only for $p = \{16, 32\}$ on the interval of $I_H \in [3.26 \ldots 3.77]$. This is consistent with our expectations; the difference between DEMO and AMS-DEMO

increases with the number of processors and is more important in the early stage of lower quality solutions and faster convergence, and less important in the later stage of higher quality solutions and slower convergence. The difference is also not detected immediately but rather after the initial random population is significantly improved.

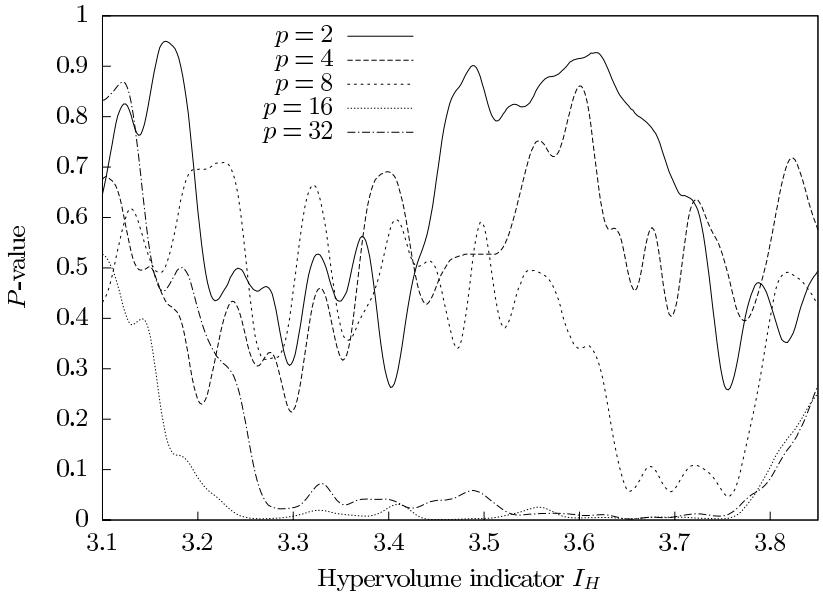

Figure 5.12: The significance of convergence rate slow down on $p > 1$ for $I_H \in [3.1 \ldots 3.85]$. The null hypotheses are that using $p \in \{2, 4, 8, 16, 32\}$ processors does not require more evaluations than using only 1 processor to reach the same value of I_H. Results plotted here are filtered with Gauss filter with $\sigma = 0.02$ to reduce the noise in P-value and thus make individual curves more clearly visible.

Convergence on the ECG Problem

The tests on the ECG problem were designed to provide more information regarding the AMS-DEMO convergence and they fulfill this goal very successfully. In the Figure 5.13, I_H is plotted against N, producing a graph very similar to the one from the tests on the cooling problem. Convergence notably slows down with increasing N, yet there is little difference between curves for $p \leq 16$. Figure 5.14 is again similar to the figure obtained from the tests on the cooling problem but with narrower confidence intervals for the mean N, thanks to a larger number of test runs. There is little visible difference in convergence rate for $p \leq 16$, but there is also a clear lowering of the convergence rate for larger values of p.

Since large number of test runs provides a very good estimate of mean N required for AMS-DEMO to reach the desired I_H value, another plot is possible. Figure 5.15 shows

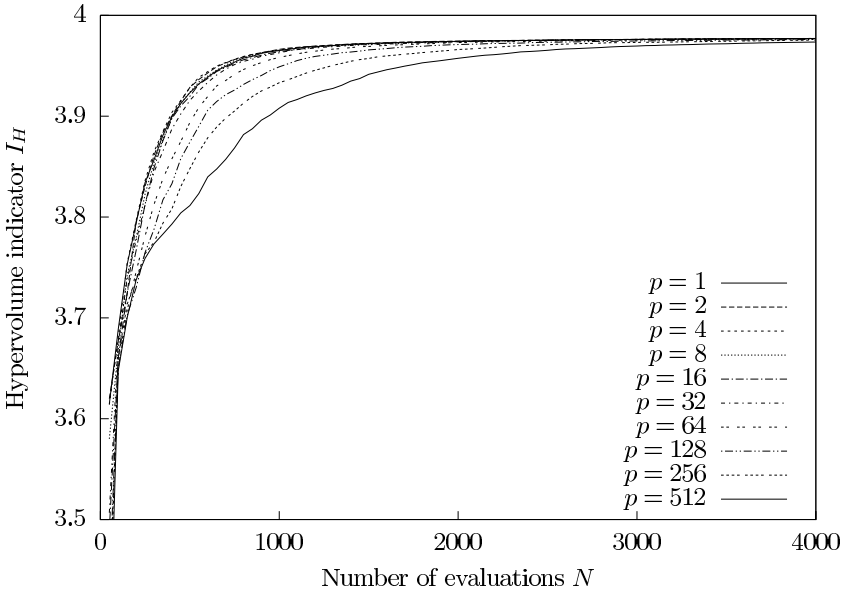

Figure 5.13: The hypervolume indicator I_H as a function of the number of evaluations N for AMS-DEMO on the ECG problem. Only up to 4000 evaluations out of 10000 are plotted as the curves visually overlap almost completely from there on.

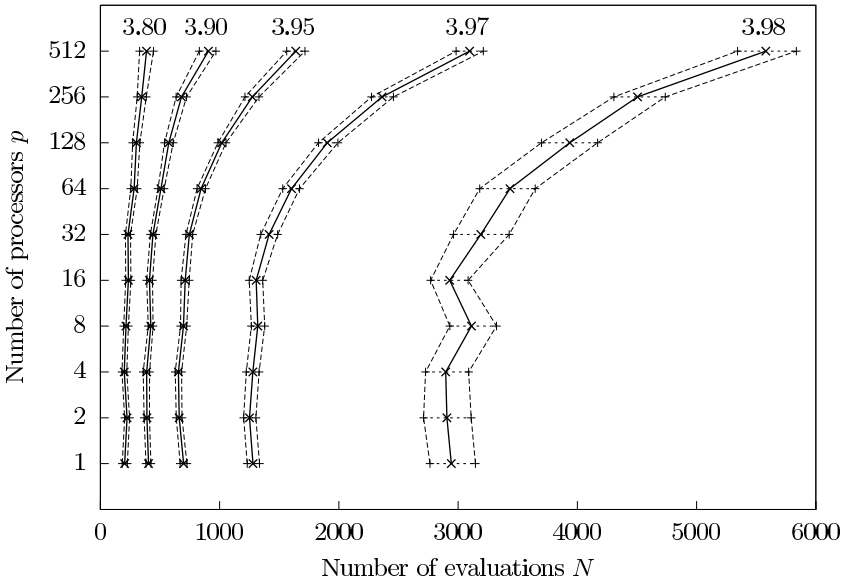

Figure 5.14: Mean number of evaluations N for AMS-DEMO to reach predefined solution quality on the ECG problem. Solution quality is specified with the hypervolume indicator I_H in labels. Best estimates for means are marked with $\times$ and connected with solid line, their 95% confidence intervals are marked with $+$, and connected with dashed lines.

the convergence rate of AMS-DEMO running on p processors relative to the original DEMO, calculated as the ratio between the mean N the compared algorithms require to

reach a certain I_H. It makes clear that the convergence rates are about constant for all p, and for AMS-DEMO at $p \leq 8$ are very similar to the convergence rate of the original DEMO (depicted by the vertical line). In contrast to previous figures, the convergence rate for $p = 16$ visibly lags behind those of smaller numbers of p and the convergence rate for $p = 4$ seems to be constantly the best, even better than the convergence rate of the original DEMO. Convergence rates of $p \geq 32$ are clearly lower than that of the original DEMO but nevertheless, the convergence rate at about 0.5 of the original convergence rate seems very good for 512 processors, which is almost 10 times the population size. A more quantitative analysis of the achieved convergence rate will be given in the next section, where speedup of AMS-DEMO running on various numbers of processors will be discussed.

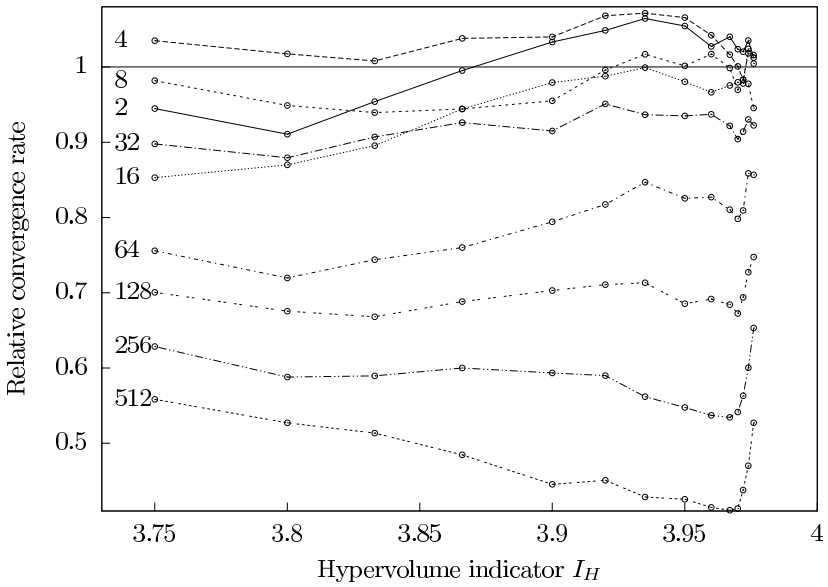

Figure 5.15: Convergence rate of AMS-DEMO on p processors (specified in labels) relative to the original DEMO on the ECG problem, as a function of the hypervolume indicator I_H. Numbers smaller than 1 indicate a convergence slower than that of the original DEMO.

Analysis of Selection Lag

So far we have been discussing convergence as a function of p, based on the dependence of the convergence rate on the selection lag l, and the linear dependence between mean l the number of processors. Since mean of l does not fully describe l, we show the distributions of l for the experimentally tested values of p. Figure 5.16 and Figure 5.17 show the distributions of l on the cooling problem for directly tested values of p and emulated values of p, respectively, while Figure 5.18 and Figure 5.19 show the same for the ECG

problem. All figures were plotted using data from log files of the master process, which contain a sequence of evaluated individuals marked with the number of the slave that evaluated them, in an order they were received and processed by the master process. All test runs, i.e. 25 per value of p for the cooling problem and 100 per value of p for the ECG problem, and all evaluated solutions within each test run were used to plot the distributions of l.

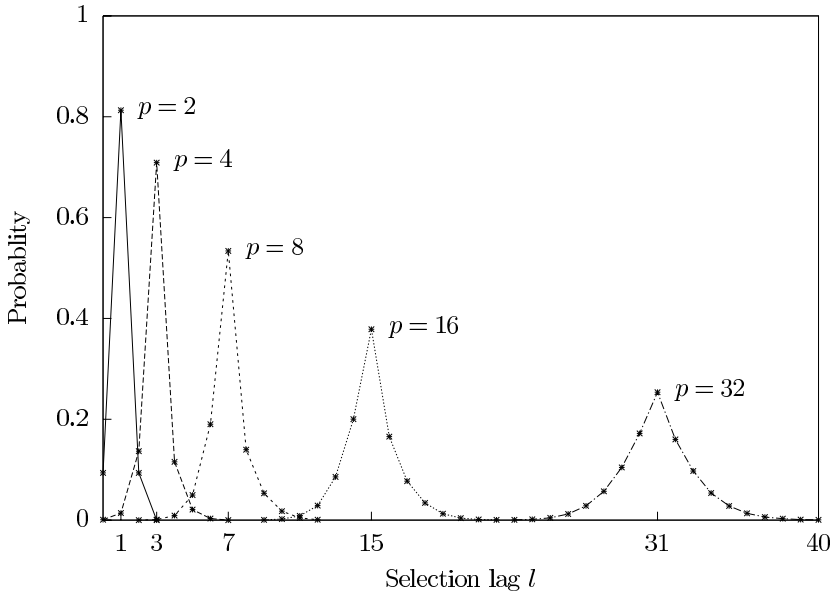

Figure 5.16: Distributions of selection lag l for various numbers of processors p on the cooling problem. Probabilities are only plotted if higher than 0.001 to limit the amount of overlap of the distribution tails.

Besides having a mean of $pq-1$, as predicted, peaks of almost all distributions of l are at $pq-1$. Exceptions are only the distributions for $p = \{320, 640\}$ on the cooling problem. These two have peaks at slightly larger values of l, because they are asymmetrical, with a fatter left tail. This is likely the result of evaluation time t_e distribution for the cooling problem. On the ECG problem the distributions of l, in contrast, reflect only very faintly the two peak nature of the distribution of t_e. Along the very pronounced peak at $l = pq-1$, there is also a weak peak at $l = 0$, almost too small to be visible in the figures.

We conclude that some real-life scenarios of non-constant evaluation time (such as the two presented in our test problems) do not have a large influence on the selection lag distribution. Observing the presented distributions of selection lag, mean is confirmed as their most informative measure. The number of processors, closely coupled to the mean selection lag, is therefore also confirmed as a good indicator of the AMS-DEMO behavior.

To summarize, the results of the convergence tests confirm the predicted lowering of

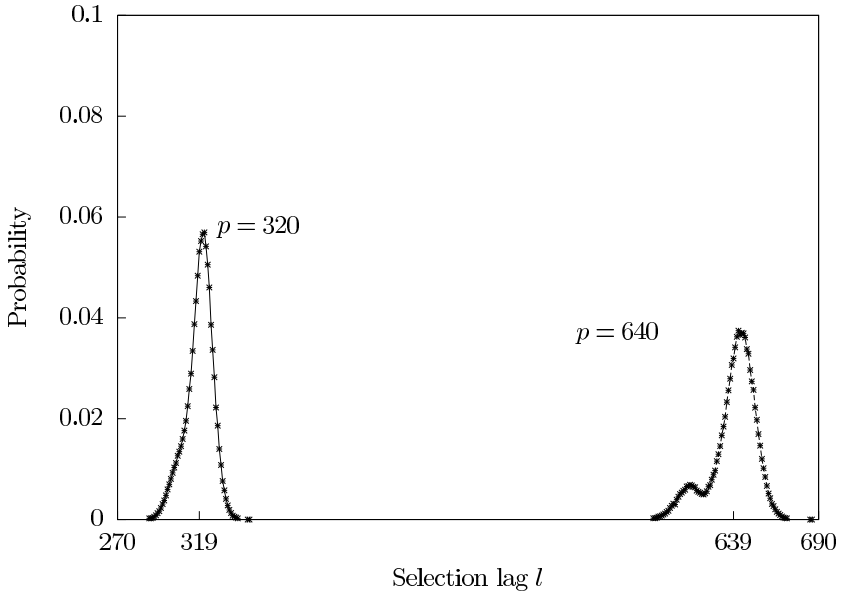

Figure 5.17: Distributions of selection lag l for tests with emulated values of number of processors p on the cooling problem. Note that y axes does not match Figure 5.16. Probabilities are only plotted if higher then 0.0002 to limit the amount of overlap of the distribution tails.

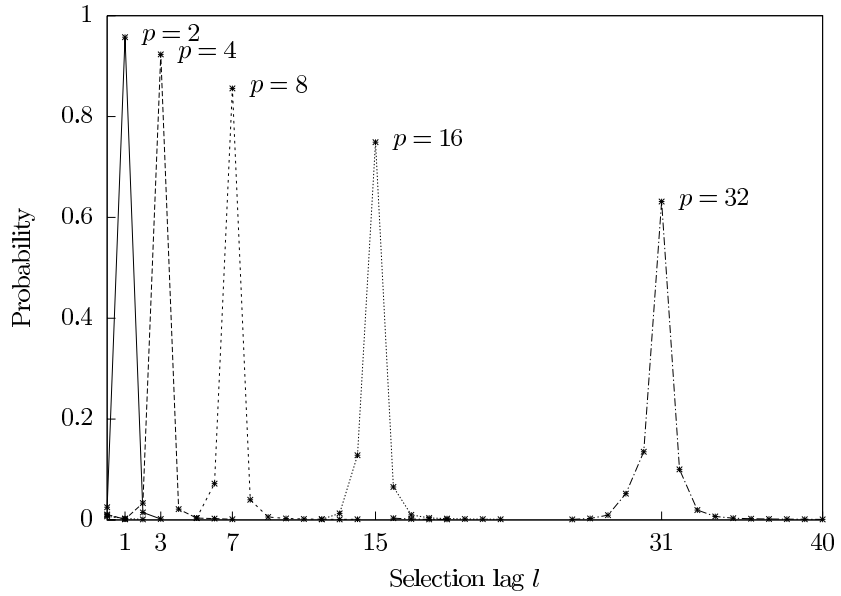

Figure 5.18: Distributions of selection lag l for various numbers of processors p on the ECG problem. Probabilities are only plotted if higher than 0.001 to limit the amount of overlap of the distribution tails.

the convergence rate of AMS-DEMO with increasing selection lag. Selection lag tracks the number of processors as predicted, meaning that the convergence rate lowers with increasing number of processors p. The limit up to which AMS-DEMO converges as fast

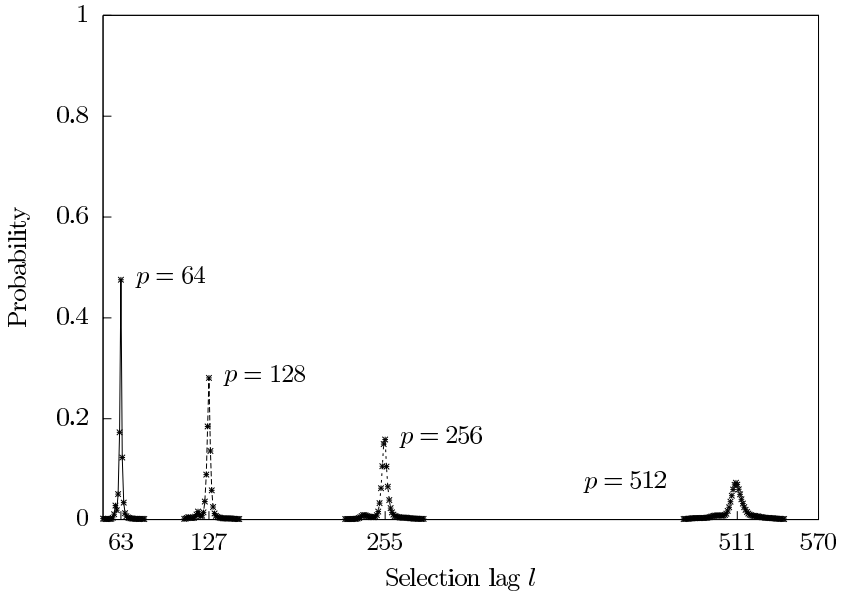

Figure 5.19: Distributions of selection lag l for tests with emulated values of number of processors p on the ECG problem. Probabilities are only plotted if higher than 0.001 to limit the amount of overlap of the distribution tails.

as the original DEMO is not clear though. On the cooling problem, the convergence of AMS-DEMO seems to slow somewhat even with p as low as 2, while on the ECG problem, the slow down only occurs at $p \geq 16$. The slow down is also quite moderate; it decreases at a slower rate than p increases, indicating the speedup will increase for the whole tested range of processors.

5.3.2 Speedup

Using speedup, we primarily address the performance of AMS-DEMO compared to the original DEMO. We also do a comparison with generational DEMO, which provides a deeper insight into the differences arising from the use of asynchronous versus synchronous parallelization. For the input data, we use the same test as for the analysis of convergence. Speedup is calculated using execution times of serial and parallel algorithms as defined by Equation 2.3. Execution times are the times in which the compared algorithms reach the same predefined quality of solutions. We measure the quality of solutions as the hypervolume indicator I_H of the nondominated set of solutions and therefore the parallel execution time of the algorithm running on p processors is referred to as $t(p, I_H)$. We can now rewrite speedup S as a function of two variables p and I_H as:

$$S(p, I_H) = \frac{t(1, I_H)}{t(p, I_H)} \ . \tag{5.1}$$

Table 5.1: The distribution of total execution times for test both problems, given as the mean ± standard deviation in seconds

p	Generational DEMO		AMS-DEMO	
	Cooling problem			
1	298502	± 1576	293667	± 3758
2	145584	± 5646	152952	± 1678
4	79751	± 446	76815	± 783
8	41105	± 389	38370	± 168
16	21454	± 183	19279	± 59
32	11019	± 276	9629	± 18
	ECG problem			
1			55720	± 326
2			27986	± 138
4			14013	± 86
8			7024	± 54
16			3504	± 36
32			1756	± 17

This equation applies only to AMS-DEMO, which is an algorithm that works differently on different numbers of processors. Speedup for generational DEMO can be calculated with basic speedup equation (Equation 2.3) because its convergence, as confirmed in previous section, is not significantly different from the convergence of the original DEMO.

To calculate the speedup, measurements of parallel execution time, also called wall clock time, are required. Parallel execution time is the time in which the algorithm in question terminates because it reaches its termination condition, and is measured from the first of the processes that starts the execution, to the last of the processes to finish the execution. For the presented parallel algorithms, the master process is both the first to start and the last to finish, therefore we measure parallel execution time as the execution time of the master process. In addition, we embed timers with the precision of $1\mu s$ into the algorithms at main points of interest – the evaluation, communication, waiting for messages, input / output operations, and DEMO operators. Although the main use of these timers is in the analysis of parallel algorithms, which is discussed in the next section, we also use them to measure execution time $t(p, I_H)$ as the time of the first population truncation which produces the population with the specified value of I_H. Table 5.1 summarizes total execution times (wall clock times) for both test problems. All times are specified in seconds, as mean value ± standard variation. Only the tests with no emulated processors are shown. Note that ECG problem is not specified in generational DEMO column as it was not solved with generational DEMO.

To aid the analysis, the speedup was broken down into two factors – the speedup due to the increased computational resources provided by the use of multiple processors, S_{p}, and the speedup due to the changes in the algorithm that were necessary for parallelization, S_{c}:

$$S(p) = S_{\mathrm{c}}(p)S_{\mathrm{p}}(p) \ . \tag{5.2}$$

The former tells us how many times more the problem related computation (overhead not included) is done per time unit on p processors in comparison to 1 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 1 processor or, in other words, how much is the convergence faster on p processors than on 1 processor. It is evident from the algorithm convergence tests that in the case of AMS-DEMO, increasing p causes an increase in computational effort (lowers the convergence rate), therefore 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:

$$S_{\mathrm{c}}(p, H) = \frac{N_q(1, I_H)}{N_q(p, I_H)} \ . \tag{5.3}$$

Finally, the speedup arising from the use of multiple processors, S_{p}, can be calculated as the ratio of the numbers of performed evaluations per time unit N_t on p processors and on 1 processor:

$$S_{\mathrm{p}}(p) = \frac{N_t(p)}{N_t(1)} \ . \tag{5.4}$$

From Table 5.2 the average difference between the AMS-DEMO and the generational DEMO can be observed on the cooling problem for $I_H \in [3.1 \ldots 3.85]$. The expected decrease in AMS-DEMO efficiency as the result of the increasing number of processors is quantified in a column for S_{c} under AMS-DEMO. AMS-DEMO therefore gets progressively less efficient than the original DEMO with every additional processor. On the other hand, a high, nearly linear S_{p} for small numbers of p implies a very good utilization of processors. The opposite holds for the generational DEMO. While it is as efficient as the original DEMO in utilizing evaluations, it is less efficient at utilizing additional processors, which is reflected in smaller S_{p}. At $p = n$, generational DEMO reaches its limit in the number of processors it can utilize, which means all further increases in p do not change its speedup, which stays at its maximum value. Note that for the calculation of speedups $S(p)$ and $S_{\mathrm{p}}(p)$ for values $p > 32$, execution times are calculated using Equation 4.3. $S_{\mathrm{c}}(p)$, on the other hand, is calculated from measured $I_H(p)$ for all p. Also note that Equation 5.2 does not hold for the table – this is because the involved quantities in table are means and not individual values, for which the equation holds.

AMS-DEMO speedups on the ECG problem are summarized in Table 5.3. As on the cooling problem, in the calculation of speedups $S(p)$ and $S_{\mathrm{p}}(p)$ for values $p > 32$, execution

Table 5.2: Comparison between AMS-DEMO and generational DEMO in terms of the mean speedup on the cooling problem for $I_H \in [3.1 \dots 3.85]$

	AMS-DEMO			Generational DEMO		
p	$S(p)$	$S_c(p)$	$S_p(p)$	$S(p)$	$S_c(p)$	$S_p(p)$
Measured						
1	**1**	1	1	**1**	1	1
2	**1.95**	0.97	2.02	**1.91**	1	1.9
4	**3.77**	0.98	3.86	**3.75**	1	3.74
8	**7.42**	0.93	8.01	**7.27**	1	7.26
16	**12**	0.78	15.4	**13.9**	1	13.9
32	**24.4**	0.80	30.6	**27.1**	1	27.1
Partially emulated						
320	**95.9**	0.36	267	**27.1**	1	27.1
640	**118**	0.29	405	**27.1**	1	27.1

Table 5.3: AMS-DEMO speedup on the ECG problem for $I_H \in [3.7 \dots 3.976]$

p	$S(p)$	$S_c(p)$	$S_p(p)$
Measured			
1	**1.000**	1.000	1.000
2	**1.966**	1.006	1.997
4	**4.116**	1.037	3.982
8	**7.580**	0.971	7.922
16	**14.17**	0.939	15.68
32	**27.36**	0.913	30.74
Partially emulated			
64	**44.95**	0.785	59.73
128	**75.35**	0.691	113.6
256	**117.8**	0.581	210.8
512	**167.6**	0.461	381.9

times are calculated using Equation 4.3, while $S_c(p)$ is calculated from measured $I_H(p)$ for all p, and Equation 5.2 does not hold because the values in the table are means.

Observing the speedups calculated from the tests on the cooling problem, the conclusion is that on homogeneous computer architectures, the algorithms closely match, with generational DEMO in a slight advantage on $p \leq n$, and AMS-DEMO in clear advantage on $p \geq n$. These tests do favor generational DEMO slightly, when using p that divides n, but it is expected that in most real-life problems n could easily be tuned to a multiple

of p, while keeping it in the interval that suits the problem. From the tests on both problems, it is clearly seen that the drop in efficiency of AMS-DEMO with increasing p is compensated by a very good utilization of available processors. On both problems, increasing p always yields greater S. In cases when p is higher than n, AMS-DEMO provides speedups far beyond the capabilities of generational DEMO.

5.4 Analytical comparison

Based on the analytical models for prediction of run time, we compare AMS-DEMO to the original DEMO and generational DEMO, providing a way to estimate the efficiency of parallel algorithms on problems and parallel architectures not tested in this thesis. Analytical model is evaluated by the timer data collected in the tests described previously.

As mentioned before, we use software timers with resolution of 1 μs to measure the duration of each repetition of the most time consuming steps of presented parallel algorithms – solution evaluation, communication, waiting for messages, input / output operations and DEMO operators together with the parallelization overhead. Evaluation time of solutions is measured on slave and master processes for generational DEMO, and only on slave processes for AMS-DEMO, whose master process does not evaluate solutions. In case of generational DEMO, the communication and waiting for messages are not separated in the source code, so the two are timed together.

We analyze the gathered time data to assure parallel algorithms behave as expected, and to contrast the processor utilization efficiency of AMS-DEMO to inefficiency of generational DEMO. First, we take a look at the separation of execution time into the main steps that are summarized for generational DEMO master in Table 5.4 and AMS-DEMO master in Table 5.5. These times confirm that the evaluation accounts for the most of the execution time in all cases. Communication and wait times associated with it only become important on multiple processors on generational DEMO. Input / output operations, comprised of algorithm reading the input parameters from a file, writing both intermediate and final results, and timer values to files, proved negligible for both algorithms. So did the DEMO operators. Further analyses can thus be simplified by ignoring the time requirements of input / output operations and DEMO operators.

Considering generational DEMO, we can understand communication times better by analyzing them per generation. In addition to the communication time and the evaluation time as measured on the master process, Figure 5.20 also shows the maximum time of all evaluations in a generation. It can be seen that the measured communication time roughly equals the difference between the longest evaluation time and the evaluation time on the master process. Measured communication time is therefore, as predicted, mostly spent waiting for the longest evaluations. Pure communication time can be estimated as the sum of communication and evaluation times on the master process, from which the longest

Table 5.4: The separation of wall clock time among the steps of the generational DEMO algorithm on the cooling problem, given as the mean ± standard deviation in seconds

	1 processor	32 processors
Total	298502 ± 1762	11019 ± 308
Evaluation and waiting	298502 ± 1762	9911 ± 376
Communication		1108 ± 129
Input/output	0.323 ± 0.012	0.307 ± 0.001
DEMO operators	0.141 ± 0.017	0.135 ± 0.003

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 variation operators, but still negligible in comparison to the evaluation time.

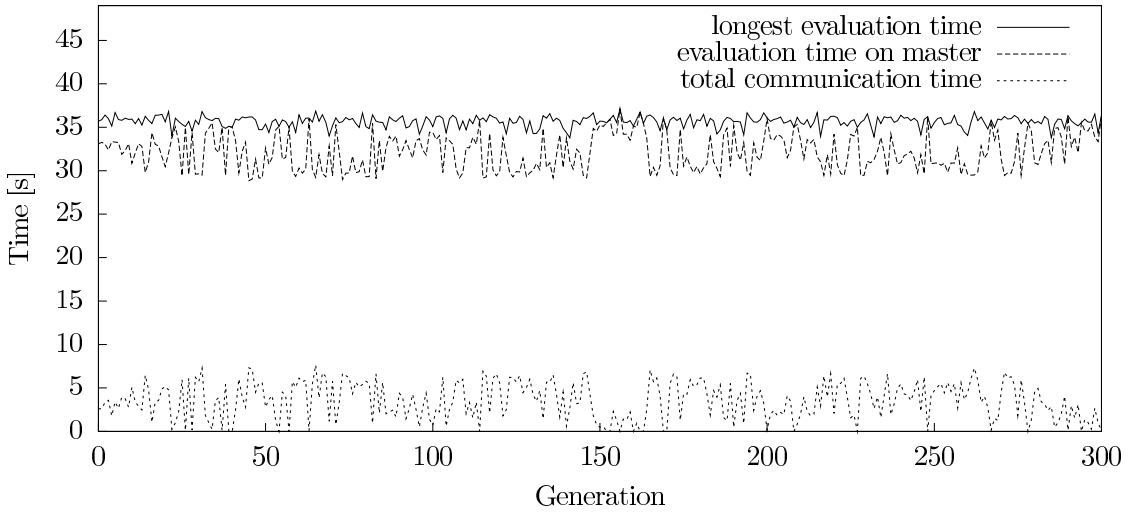

Figure 5.20: Single run of evaluation and interprocess communication times per generation of generational DEMO on the cooling problem on 32 processors. Evaluation time of the master process is contrasted with the longest evaluation time of all the processes. The difference between the these times matches closely the communication time.

Master process of AMS-DEMO spends most of the time idle, but as described before, this has no effect on the algorithm performance, since the master always shares processor with one slave. In comparison to generational DEMO, input / output operations in AMS-DEMO have a greater time complexity that also varies with the number of processors, on the account of writing additional debugging information, spread up among the proces-

Table 5.5: The separation of the wall clock time among the steps of the AMS-DEMO algorithm on the ECG problem

	1 processor	32 processors
Total	55273 ± 286	1747 ± 17
Communication		2.48 ± 0.13
Idle	55269 ± 286	1739 ± 17
Input/output	2.71 ± 0.74	1.61 ± 0.07
DEMO operators and overhead	1.48 ± 0.02	4.02 ± 0.14

sors. Similarly, for AMS-DEMO, the time complexity of DEMO operators and overhead is larger and increases with the number of processors. The parallelization overhead, comprised of working with local copies of queues and storing links between parents and offspring, can be assumed to cause this variability. Unfortunately it is tightly interwoven with the DEMO operators, which makes it impractical to measure them separately.

The presented time measurements confirm that estimations of the execution time in Equation 4.1 and Equation 4.3 were constructed correctly, using only the most time demanding steps of both parallel algorithms. To establish the accuracy of the equations and possibility to use them for prediction of behavior of the presented algorithms, the estimated execution times are compared against the experimentally measured execution times in calculation of the speedup due to use of additional processors, S_p.

Estimations of the execution time require the distributions of evaluation time to be specified. Therefore, the distributions are estimated from the measurements for both test problems. We use the measurements made by $1\mu s$ resolution timer that measures every evaluation separately. The results are presented in Figure 5.21 for the cooling problem and in Figure 5.22 for the ECG problem. Both problems exhibit some variability in their evaluation time, and the distribution of the ECG problem evaluation time consists of two separate peaks at 0 s and at 5.5 s, which are the consequence of a two stage evaluation, where the second, more time demanding stage, is not always executed. There is also a part of the distribution to the right of the plotted area of t_e for the ECG problem (7183 samples out of 1000000) consisting of very long evaluations, lasting from 6.25 s to 320 s. These long evaluations are caused by the hardware taking several orders of magnitude longer to calculate exponential function x^y observed for some rare combinations of arguments x and y than it does for others. Furthermore, the distribution of evaluation times on the ECG problem contains a time dimension as well – the number of extremely bad solutions that evaluate extremely fast decreases as the algorithm converges, as we have seen in Subsection 5.2.2. We make an additional plot of the number of evaluations that last less than a second (fast evaluations) relative to the hypervolume indicator I_H of the nondominated set of solutions in Figure 5.23. The presented decrease in the number of

fast evaluations with the increasing I_H is not of much use for prediction of run times though, because the convergence of I_H is not known beforehand.

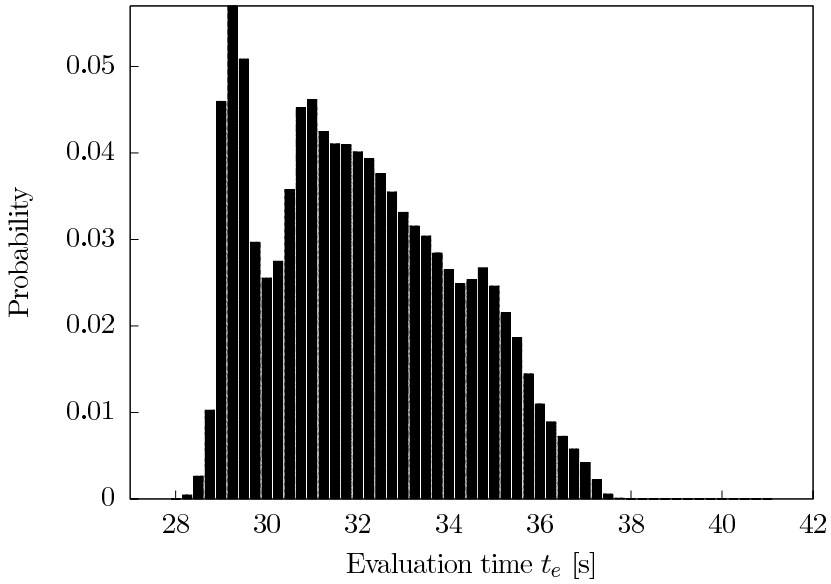

Figure 5.21: Distributions of evaluation times for the cooling problem made by timing 240000 evaluations, that were performed during the test runs on 32 processors.

Then, using the obtained evaluation time distributions, the estimated and measured S_p are calculated and plotted in Figure 5.24 and Figure 5.25. Both figures show accurate estimations of speedup, even though the evaluation time variability in dependence of I_H is ignored in our runtime estimations for the ECG problem. There are slight errors between the predicted and measured speedups of the generational DEMO that occur because of the small number of experimental runs, that do not smooth out variations in t_e completely.

5.5 Test on a heterogeneous computer architecture

To test the AMS-DEMO flexibility, a heterogeneous computer architecture has been created for experimental optimization of the cooling problem. Two computers from the cluster described in Section 5.1 and two desktop computers have been connected via a Gigabit Ethernet switch, creating a six processor architecture. Details of the heterogeneous architecture along with the statistics of a single AMS-DEMO run are summarized in Table 5.6. Total execution time of the run equals 49407 s. Aside from the differences in the declared processor clock frequencies, the computers also feature different operating systems, and in the table, the two desktop computers are named according to these. Al-

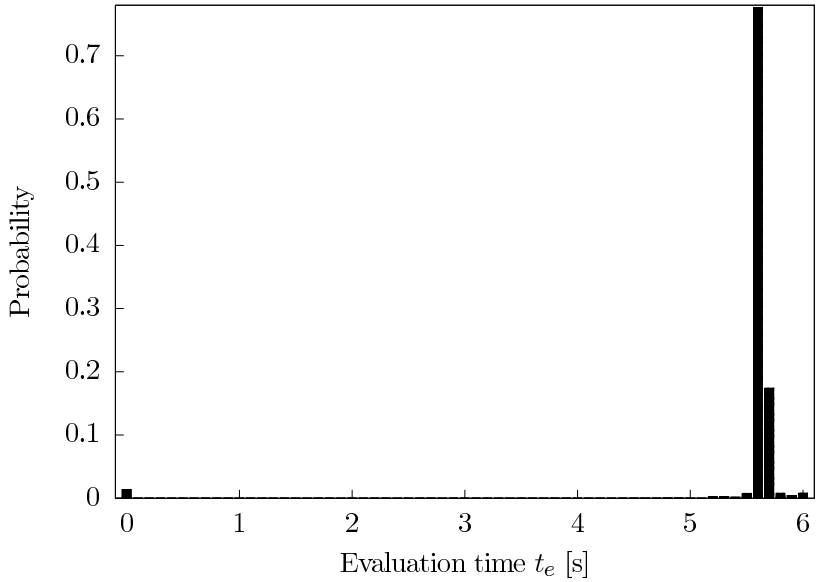

Figure 5.22: Distributions of evaluation times for the and ECG problem made by timing 1000000 evaluations, that were performed during the test runs on 32 processors.

though different mean evaluation times among the computers have been expected when setting up the architecture, the additional nondeterministic variation in evaluation times were found, which differed between the computers as shown in Figure 5.26. This nondeterministic behavior was not explored any further but it should serve as a warning that some nondeterminism in evaluation times can be expected on real computer systems and real optimization problems.

The results from the heterogeneous architecture obtained by averaging five AMS-DEMO runs are compared against the estimated performance of generational DEMO on the same computer architecture. Two possible settings of the generational DEMO are considered, that differ only in the population size. The first setting uses the population size of 32, the same as used in the AMS-DEMO tests, which is not a multiple of the number of processors, and therefore causes a degradation of performance of the generational DEMO. For the second setting, the population size is adjusted to the nearest multiple of the number of processors, which is 30. The performance of the generational DEMO was estimated using Equation 4.1 and the distributions of the evaluation times from Figure 5.26. Note that we decided to calculate execution times instead of measuring them because the experiments requiring dedicating desktop computers for this task for extended periods of time were impractical.

In Table 5.7 the performance of AMS-DEMO and generational DEMO on the heterogeneous architecture are summarized. It follows that AMS-DEMO has a significant advantage over generational DEMO on the presented architecture in processor utilization, resulting in the processors being idle for only a negligible 14 seconds on average. On

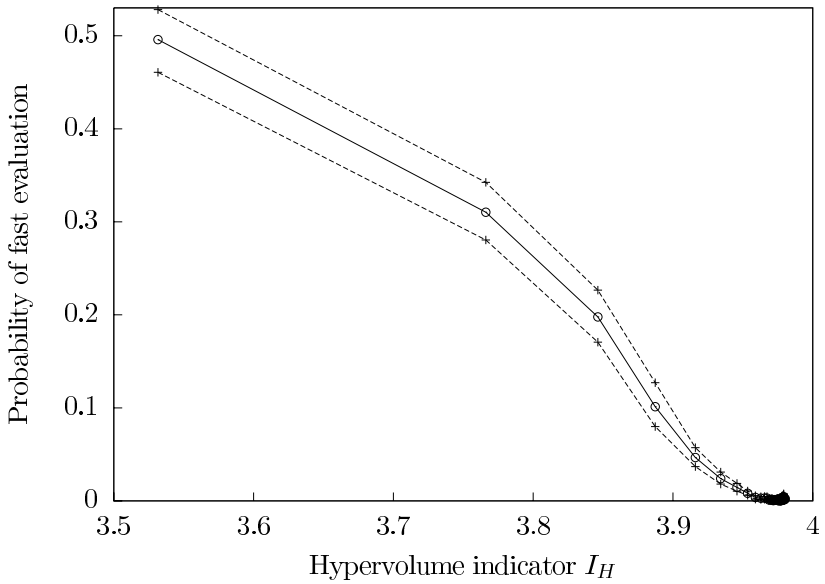

Figure 5.23: Probability of generating a solution from a population with a given I_H which will evaluate in less than 1s (will only go through the first stage of the evaluation) on the ECG problem, taken from 100 runs of AMS-DEMO on 32 processors (total of 1000000 evaluations). Mean values are plotted in circles and solid line, while the confidence intervals of the mean at 0.99 confidence are plotted in crosses and dashed lines. Each point represents a mean of 10000 evaluations therefore there is a total of 100 points plotted, most of them grouped on the right of the graph, because most of the evaluations is typically done when I_H of the active population is high.

the account of long idle times of processors, the wall clock times of generational DEMO algorithm are 21 % or 11 % longer, depending on whether we use population size of 30 or 32, than those of AMS-DEMO for the same number of evaluations. AMS-DEMO remains faster even when also consider the S_c, which equals 1 for generational DEMO and lies between 0.93 and 0.98 for AMS-DEMO solving the ECG problem on 6 processors (estimated from rows for 4 and 8 processors in Table 5.2). This means AMS-DEMO requires between 2 % and 7 % more evaluations than generational DEMO for the similar quality solutions – not enough to offset the much shorter wall clock time.

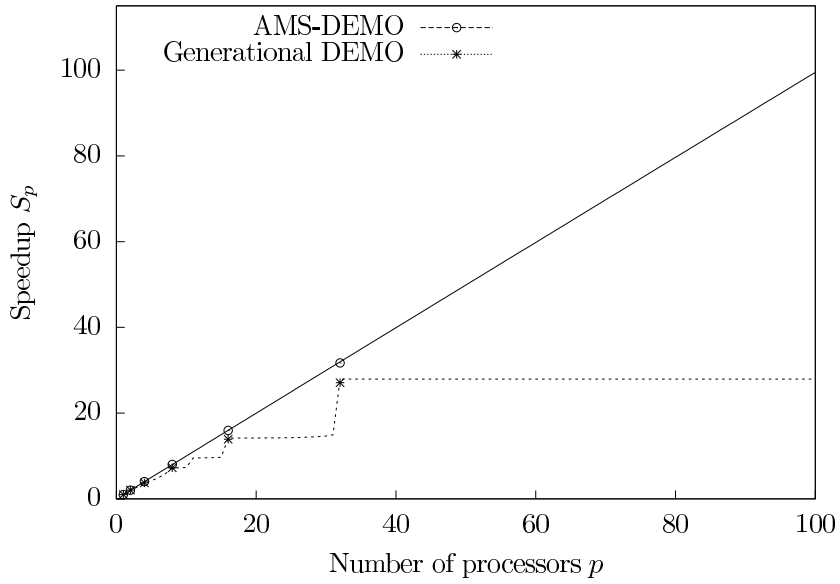

Figure 5.24: Estimated (lines) and measured (markers) speedups S_p for generational DEMO and AMS-DEMO on the cooling problem.

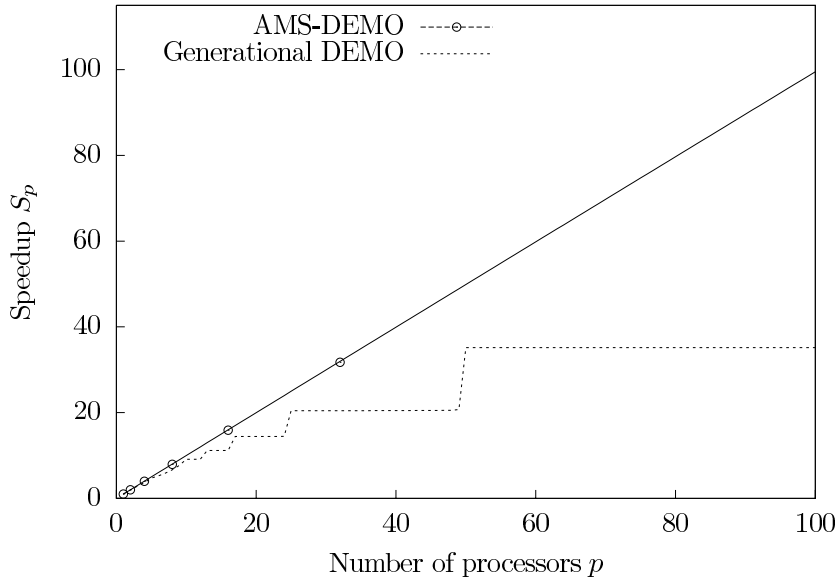

Figure 5.25: Estimated (lines) and measured (markers) speedups S_p for generational DEMO and AMS-DEMO on the ECG problem. Note that there are no measured speedups marked for generational DEMO because generational DEMO was not applied on the ECG problem.

Table 5.6: Technical data and performance of the heterogeneous computer architecture

	Cluster	Windows PC	Linux PC
Number of processors	4	1	1
Operating system	64 bit Linux	32 bit Windows	64 bit Linux
Processor type	Opteron 244	Athlon 64	Athlon 64
Clock frequency	1800 MHz	2200 MHz	2000 MHz
Number of evaluations per processor	1551 1551 1537 1540	1702	1719
Mean evaluation time	31.98 s	29.01 s	28.73 s

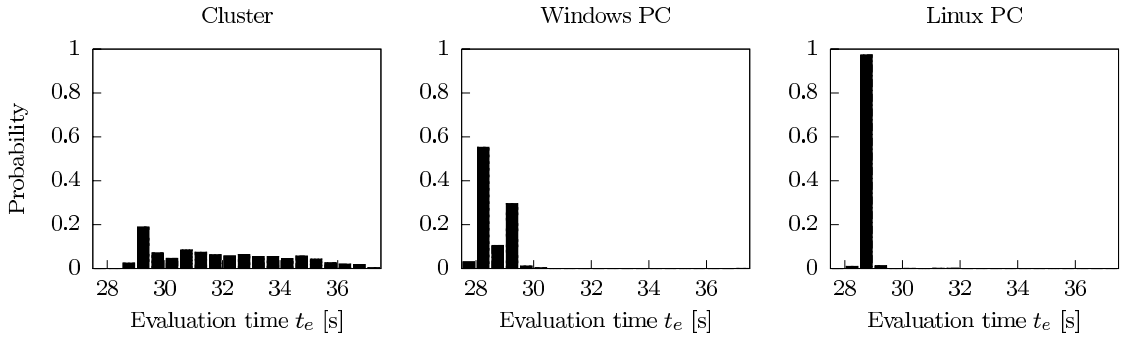

Figure 5.26: Probability distribution of the evaluation time on the cooling problem on computers from the heterogeneous test architecture.

Table 5.7: Comparison of algorithm performances on the heterogeneous computer architecture

	AMS-DEMO	Generational DEMO	
Population size	32	32	30
Wall clock time	49669 s	60176 s	54939 s
Mean processor idle time	14 s	11077 s	5625 s
Idle time ratio	0.028 %	18.4 %	10.2 %

Chapter 6

Conclusions and Further Work

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). The 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 characterizes this change entirely – was identified. Selection lag depends linearly on the number of processors and queue sizes, and has an adverse effect on the algorithm, increasing the number of evaluations required to find high quality solutions. Tests on real-world problems indicate the effect of selection lag is hardly noticeable for selection lags lower than about half the population size. Only for larger selection lags 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 additional processors, resulting in shorter optimization times and larger speedups. This property is robust, and holds for all the performed tests, even with the number of processors up to several times the population size.

The constraints for the number of processors were also relaxed, compared to the constraints imposed by the synchronous master-slave parallelization. The number of processors does not require to divide the population size and may even exceed it. The asynchronous master-slave parallelism also allows for dynamically changing number of processors, whether it be intentional or accidental – which makes it robust in error handling, and convenient for use even on computer architectures with dynamically changing

load or with unreliable communication networks.

We tested the AMS-DEMO algorithm on two multiobjective optimization problems. Both problems were solved successfully and in much shorter time than by the serial algorithm. Therefore, the parallelization of the optimization algorithm was used to make optimization more manageable while in the future it might serve to support solving more demanding versions of the same problems. The efficiency of the proposed AMS-DEMO algorithm was contrasted against a simpler and more straight-forward parallelization in which the DEMO algorithm had first been made generational and then parallelized using synchronous master-slave parallelization method. The tests 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 the conditions unfavorable to synchronous parallelism accumulate, however, the AMS-DEMO gains advantage, as the test on a heterogeneous architecture shows. We conclude that when the conditions are favorable – population size is a multiple of the number of processors, evaluation time is near constant, and architecture is homogeneous – a simpler synchronous master-slave parallelization could still be preferred over the more complex asynchronous master-slave; otherwise the latter delivers much greater speedups.

Although the predictions made by the analysis and the test results so far agree, the AMS-DEMO should be further tested on other problems before making firm conclusions. Since the parallelization properties of the 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 exploring the proposed parallelization type independently of the base algorithm it is applied on. 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 of further work.

Bibliography

[1] Abbass H.; Sarker R.; Newton C. PDE: A pareto-frontier differential evolution approach for multi-objective optimization problems. In: *Proceedings of the 2001 Congress on Evolutionary Computation – CEC'01*, volume 2, 971–978. (2001).

[2] Abbass H. A. The self-adaptive pareto differential evolution algorithm. In: *Proceedings of the 2002 Congress on Evolutionary Computation – CEC'02*, volume 1, 831–836. (2002).

[3] Abraham A.; Jain L.; Goldberg R. (ed.). *Evolutionary Multiobjective Optimization*. (Springer-Verlag, London, 2005).

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

[5] Alba E.; Nebro A. J.; Troya J. M. Heterogeneous computing and parallel genetic algorithms. *Journal of Parallel and Distributed Computing* **62(9)**, 1362–1385, (2002).

[6] Alba E.; Troya J. M. Analyzing synchronous and asynchronous parallel distributed genetic algorithms. *Future Generation Computer Systems* **17(4)**, 451–465, (2001).

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

[8] Almasi G. S.; Gottlieb A. *Highly parallel computing*. (Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1989).

[9] Auger A.; Bader J.; Brockhoff D.; Zitzler E. 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*, 87–102. (ACM, New York, NY, USA, 2009).

[10] Avbelj V.; Trobec R.; Geršak B.; Vokač D. Multichannel ECG measurement system. In: *Proceedings of the 10th IEEE Symposium on Computer-Based Medical Systems, IEEE Computer*, 81–84. (Society Press, 1997).

[11] Branke J.; Schmeck H.; Deb K.; Reddy S M. Parallelizing multi-objective evolutionary algorithms: Cone separation. In: *Proceedings of the 2004 Congress on Evolutionary Computation – CEC'04*, 1952–1957. (IEEE Press, Portland, Oregon, USA, 2004).

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

[13] Cheang S. M.; Leung K.-S.; Lee K.-H. Genetic parallel programming: Design and implementation. *Evolutionary Computation* **14(2)**, 129–156, (2006).

[14] Coello C. A. C. A Comprehensive Survey of Evolutionary-Based Multiobjective Optimization Techniques. *Knowledge and Information Systems* **1(3)**, 129–156, (1999).

[15] Coello C. A. C. Recent Trends in Evolutionary Multiobjective Optimization. In: *Evolutionary Multiobjective Optimization*, 7–23. (Springer Berlin Heidelberg, 2006), (2006).

[16] Coello C. A. C.; Lamont G. B.; Veldhuizen D. A. V. *Evolutionary Algorithms for Solving Multi-Objective Problems (Genetic and Evolutionary Computation)*. (Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006).

[17] De Jong K. *Evolutionary Computation: A Unified Approach*. (The MIT Press, Cambridge, 2006).

[18] de Toro Negro F.; Ortega J.; Ros E.; Mota S.; Paechter B.; Martín J. M. Psfga: Parallel processing and evolutionary computation for multiobjective optimisation. *Parallel Computing* **30(5-6)**, 721–739, (2004).

[19] Deb K. *Multi-Objective Optimization using Evolutionary Algorithms*. (John Wiley & Sons, Chichester, UK, 2001).

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

[21] Deb K.; Tiwari S. Omni-optimizer: A procedure for single and multi-objective optimization. In: Coello C. A. C.; Aguirre A. H.; Ziztler E. (ed.), *Proceedings of the Third Conference on Evolutionary Multi-Criterion Optimization – EMO 2005*, Lecture Notes in Computer Science, 47–61. (Springer, 2005).

[22] Deb K.; Zope P.; Jain A. Distributed computing of pareto-optimal solutions with evolutionary algorithms. In: Thiele L. (ed.), *Proceedings of the Second Conference on Evolutionary Multi-Criterion Optimization – EMO 2003*, Lecture Notes in Computer Science, 534–549. (Springer, 2003).

[23] Depolli M.; Avbelj V.; Trobec R. Solving inverse problem with genetic algorithm: ECG analysis. In: *Proceedings of the Sixteenth International Electrotechnical and Computer Science Conference – ERK 2007*, volume B, 69–72. (2007).

[24] Depolli M.; Avbelj V.; Trobec R. Computer-simulated alternative modes of U-wave genesis. *Journal of Cardiovascular Electrophysiology* **19(1)**, 84–89, (2008).

[25] Depolli M.; Erkki Laitinen B. F. Parallel differential evolution for simulation-based multiobjective optimization of a production process. In: Filipič B.; Šilc J. (ed.), *Proceedings of the Fourth International Conference on Bioinspired Optimization Methods and their Applications – BIOMA 2010*, 141–152. (2010).

[26] Depolli M.; Tušar T.; Filipič B. Tuning parameters of a multiobjective optimization evolutionary algorithm on an industrial problem. In: *Proceedings of the Fifteenth International Electrotechnical and Computer Science Conference – ERK 2006*, volume B, 95–98. (2006).

[27] di Bernardo D.; Murray A. Origin on the electrocardiogram of U-waves and abnormal U-wave inversion. *Cardiovascular Research* **53(1)**, 202–208, (2002).

[28] Eiben A. E.; Smith J. E. *Introduction to Evolutionary Computing*. (Springer-Verlag, Berlin, 2003).

[29] Filipič B.; Depolli M. Parallel evolutionary computation framework for single- and multiobjective optimization. In: Trobec R.; Vajterčic M.; Zinterhof P. (ed.), *Parallel Computing – Numerics, Applications, and Trends*, 217–240. Springer, Dordrecht, (2009).

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

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

[32] Flynn M. J. Some computer organizations and their effectiveness. *Transactions on Computers* **21(9)**, 948–960, (1972).

[33] Fox G. C.; Johnson M. A.; Lyzenga G. A.; Otto S. W.; Salmon J. K.; Walker D. W. *Solving problems on concurrent processors. Vol. 1: General techniques and regular problems*. (Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988).

[34] Ghazali Talbi E.; Meunier H. Hierarchical parallel approach for GSM mobile network design. *Journal of Parallel and Distributed Computing* **66(2)**, 274–290, (2006).

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

[36] Heermann D. W.; Burkitt A. N. *Parallel algorithms in computational science*. (Springer-Verlag New York, Inc., New York, NY, USA, 1991).

[37] Herrero J. M.; Blasco X.; Martinez M.; C. Ramos J. S. Non-linear robust identification using evolutionary algorithms: Application to a biomedical process. *Engineering Applications of Artificial Intelligence* **21(8)**, 1397–1408, (2008).

[38] Hiroyasu T.; Miki M.; Watanabe S. Parallel evolutionary optimization of multibody systems with application to railway dynamics. *Multibody System Dynamics* **9(2)**, 143–164, (2003).

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

[40] Kumar V.; Grama A.; Gupta A.; Karypis G. *Introduction to parallel computing: Design and analysis of algorithms*. (Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994).

[41] Liu P.; Kang L.; de Garis H.; Chen Y. An asynchronous parallel evolutionary algorithm (apea) for solving complex non-linear real world optimization problems. *Neural, Parallel and Scientific Computations* **10(2)**, 179–188, (2002).

[42] Luna F.; Nebro A. J.; Alba E. Parallel Evolutionary Multiobjective Optimization. In: *Parallel Evolutionary Computations*, 33–56. (Springer Berlin, Heidelberg, 2006), (2006).

[43] Macfarlane P. W.; Lawrie T. D. V. (ed.). *Comprehensive Electrocardiology: Theory and Practice in Health and Disease*, volume 1. (Pergamon Press, New York, 1st edition, 1989).

[44] Mai G. C.; F C. A.; Rose D. Low cost cluster architectures for parallel and distributed processing, (2000).

[45] Nebro A. J.; Luna F.; Talbi E.-G.; Alba E. Parallel multiobjective optimization. In: Alba E. (ed.), *Parallel Metaheuristics*, 371–394. John Wiley & Sons, New Jersey, (2005).

[46] Oliveira L. S.; R.Sabourin ; Bortolozzi F.; Suen C. 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, (2003).

[47] Parsopoulos K. E.; Tasoulis D. K.; Pavlidis N.; Plagianakos V. P.; Vrahatis M. N. Vector evaluated differential evolution for multiobjective optimization. In: *2004 Congress on Evolutionary Computation (CEC 2004)*, 204–211. (2004).

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

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

[50] Quagliarella D.; Vicini A. 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*, 3142–3147. (1998).

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

[52] Ritsema van Eck H. J.; Kors J. A.; van Herpen G. The U wave in the electrocardiogram: A solution for a 100-year-old riddle. *Cardiovascular Research* **67(2)**, 256–262, (2005).

[53] Ritsema van Eck H. J.; Kors J. A.; van Herpen G. Dispersion of repolarization, myocardial iso-source maps, and the electrocardiographic T and U waves. *Journal of Electrocardiology* **39(4 Suppl)**, 96–100, (2006).

[54] Robič T. Performance of DEMO on new test problems: A comparison study. In: *Proceedings on the 14th International Electrotechnical and Computer Science Conference – ERK 2005*, volume B, 121–124. (2005).

[55] Robič T.; Filipič B. 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*, 520–533. (Springer, 2005).

[56] Rowe J.; Vinsen K.; Marvin N. Parallel GAs for multiobjective functions. In: *Proceedings of the Second Nordic Workshop on Genetic Algorithms and Their Applications – 2NWGA*, 61–70. (1996).

[57] Silverman B. W. *Density estimation for statistics and data analysis*. (Chapman and Hall, London, 1986).

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

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

[60] Streichert F.; Ulmer H.; Zell A. Parallelization of multi-objective evolutionary algorithms using clustering algorithms. In: Coello C. A. C.; Aguirre A. H.; Ziztler E. (ed.), *Proceedings of the Third Conference on Evolutionary Multi-Criterion Optimization – EMO 2005*, volume 3410 of *Lecture Notes in Computer Science*, 92–107. (Springer, 2005).

[61] Ten Tusscher K.; Noble D.; Noble P.; Panfilov A. A model for human ventricular tissue. *American Journal of Physiology: Heart and Circulatory Physiology* **286(4)**, H1573–H1589, (2004).

[62] Trobec R. Two-dimensional regular d-meshes. *Parallel computing* **26(13–14)**, 1945–1953, (2000).

[63] Trobec R.; Depolli M.; Avbelj V. ECG simulation with improved model of cell action potentials. In: *HEALTHINF*, 18–21. (2009).

[64] Trobec R.; Depolli M.; Avbelj V. Simulation of ECG repolarization phase with improved model of cell action potentials. In: Fred A.; Filipe J.; Gamboa H. (ed.), *Biomedical engineering systems and technologies : International Joint Conference BIOSTEC 2009*, 325–332. Springer, Berlin, (2010).

[65] Trobec R.; Jerebič I.; Janežič D. Parallel algorithm for molecular dynamics integration. *Parallel computing* **19(9)**, 1029–1039, (1993).

[66] Tušar T. Design of an algorithm for multiobjective optimization with differential evolution. M.Sc. thesis, Faculty of Computer and Information Science, University of Ljubljana, (2007).

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

[68] Wohlfart B. A simple model for demonstration of SST-changes in ECG. *European Heart Journal* **8**, 409–416, (1987).

[69] Xiong S.; Li F. Parallel strength pareto multiobjective evolutionary algorithm. In: *Proceedings of the Fourth International Conference on Parallel and Distributed Computing, Applications and Technologies – PDCAT'2003*, 681–683. (2003).

[70] Zitzler E. *Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications*. PhD thesis, Swiss Federal Institute of Technology Zurich, (1999).

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

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

[73] Zitzler E.; Thiele L. Multiobjective optimization using evolutionary algorithms – a comparative case study. In: *Proceedings of the Fifth Conference on Parallel Problem Solving from Nature – PPSN V*, 292–301. (Springer, Heidelberg, 1998).

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

List of Figures

List of Tables

List of Algorithms

Appendices

Appendix 1: Publications Related to This Thesis

The conditions for defense of the thesis were fulfilled by the following scientific publications:

Journal Papers

- Depolli M.; Avbelj V.; Trobec R. Computer-simulated alternative modes of U-wave genesis. *Journal of Cardiovascular Electrophysiology* **19(1)**, 84–89, (2008). IF(2008)=3.798

Book chapters

- Filipič B.; Depolli M. Parallel evolutionary computation framework for single- and multiobjective optimization. In: Trobec R.; Vajterčic M.; Zinterhof P. (ed.), *Parallel Computing*, 217–240. Springer, London, (2009).

- Trobec R.; Depolli M.; Avbelj V. Simulation of ECG repolarization phase with improved model of cell action potentials. In: Fred A.; Filipe J.; Gamboa H. (ed.), *Biomedical engineering systems and technologies : International Joint Conference BIOSTEC 2009*, 325–332. Springer, Berlin, (2010).

Proceedings papers

- Depolli M.; Tušar T.; Filipič B. Tuning parameters of a multiobjective optimization evolutionary algorithm on an industrial problem. In: *Proceedings of the Fifteenth International Electrotechnical and Computer Science Conference – ERK 2006*, volume B, 95–98. (2006).

- Depolli M.; Avbelj V.; Trobec R. Solving inverse problem with genetic algorithm: Ecg analysis. In: *Proceedings of the Sixteenth International Electrotechnical and Computer Science Conference – ERK 2007*, volume B, 69–72. (2007).

- Depolli M.; Erkki Laitinen B. F. Parallel differential evolution for simulation-based multiobjective optimization of a production process. In: Filipič B.; Šilc J. (ed.), *Proceedings of the Fourth International Conference on Bioinspired Optimization Methods and their Applications – BIOMA 2010*, 141–152. (2010).