Doctoral Thesis

DISTRIBUTED EVOLUTIONARY ALGORITHMS

Miroslav Červenka

Tomas Bata University in Zlín

Faculty of Applied Informatics

Zlín, 2006

Study program:	2807	Chemical and processing engineering
Specialization:	26-15-9	Technical cybernetics

Supervisor: doc. Ing. Ivan Zelinka, Ph.D.

CONTENTS

ACKNOWLEDGEMENTS........3
ABSTRACT........5
ABSTRAKT........6
1 AIMS OF WORK........7
2 INTRODUCTION........8
3 STATE OF THE ART........10
3.1 Parallel genetic algorithm........10
3.2 Classification of Parallel Genetic Algorithms........11
3.3 History of Parallel Genetic Algorithms........14
3.4 Master-Slave Parallel GA........15
3.5 Multiple-Population Parallel GA........17
3.6 Coarse-Grained parallel GA........19
3.7 Fine-Grained parallel GA........20
3.8 Communication topologies........20
3.9 Hierarchical parallel GA........21
3.10 Applications of parallel GAs........23
4 PLATFORM FOR PARALLEL COMPUTATIONS........25
4.1 Brief history of the platform........26
4.2 General cluster description........27
4.3 Database structure........28
4.4 Structure and functions of the platform........30
4.4.1 Package cluster.server........30
4.4.2 Package cluster.terminal........34
4.4.3 Package cluster.shared........38
4.4.4 Package cluster.utils........40
4.5 ClusterGuruGUI........41
4.6 Cluster configuration file........44
4.7 How to write a cluster application........45
4.7.1 Example #1 – Primes........45
4.7.2 Example #2 – TimeCluster........49
4.8 Cluster performance........51
4.8.1 TimeCluster........52
4.8.2 PrimeCluster........53

4.8.3 DataCluster.....53
4.9 Chapter Summary.....53
5 PARALLEL EVOLUTIONARY ALGORITHMS – DE & SOMA.....54
5.1 Differential evolution.....54
5.1.1 Serial DE.....54
5.1.2 Parallel DE.....55
5.2 Self-Organising Migrating Algorithm.....59
5.2.1 Serial SOMA.....59
5.2.2 Parallel SOMA.....60
5.2.3 Parallel SOMA performance.....64
5.3 Chapter Summary.....67
6 APPLICATIONS.....68
6.1 Combustion engine optimization.....68
6.2 Relay node placement in energy-constrained networks.....72
6.2.1 Network model and balanced data gathering as a flow LP.....72
6.2.2 The effect of relay nodes.....74
6.3 Aerodynamic optimisation of wing geometry.....80
6.3.1 The vortex system theory.....80
6.3.2 The elliptic distribution of lift.....83
6.3.3 The general distribution of lift.....85
6.3.4 Aerodynamic characteristics for symmetrical general loading.....87
6.3.5 Determination of the load distribution on a wing.....89
6.3.6 Optimised wing model.....92
6.3.7 Optimisation results.....94
6.4 Chapter Summary.....102
7 CONCLUSIONS.....103
REFERENCES.....106
LIST OF FIGURES.....112
LIST OF TABLES.....114
LIST OF PUBLICATIONS.....115
CURRICULUM VITAE.....116

ACKNOWLEDGEMENTS

My deepest gratitude is due to my supervisor doc. Ing. Ivan Zelinka, Ph.D., who made me interested in the breathtaking world of evolutionary algorithms and their applications. Hereby I would like to express my appreciation for being a patient advisor, for supporting me with constructive criticism and new ideas.

As part of work has been done during the visit to Laboratory for Theoretical Computer Science, Helsinki University of Technology, I also would like to express my most sincere gratitude to Professor Pekka Orponen for his invitation, support and all ideas that contributed to the relay node placement optimisation project.

Special thanks go to Pavel Růžička, an aerodynamicists from the Evektor company, for his valuable assistance and support he provided me during my attachment in Evektor, where we created the aerodynamic model of a wing, whose parameters were subject to optimisation.

I am much obliged to Markéta Červinková and Zuzana Pátíková who revised the language of this manuscript.

Finally, I am forever indebted to my parents for their understanding, endless patience and encouragement when it was most required.

There must be a couple of important people I have not thanked yet – thank you very much!

Think parallel!

ABSTRACT

This work deals with the idea of parallelisation of evolutionary algorithms with the primary aim to develop a parallel version of the Self-Organising Migrating Algorithm (SOMA).

In the first part we present an overview of history and current state on the field of distributed computations and parallel evolutionary algorithms. This includes classification of parallel genetic algorithms, description of parallelisation strategies and a summary of successful applications.

Then, a fully scalable framework for parallel/distributed applications is introduced and comparison of different parallelisation strategies on this platform is described. This platform represents universal, very efficient and easily configurable framework which can be run on various platforms. It enables utilisation of tens, hundreds or even thousands of CPUs which normally idle in numerous common office computers in companies or at universities.

Parallelisation schemes of Differential Evolution (DE) and SOMA are analysed and implementation of both algorithms is extensively described. Four different parallelisation approaches of SOMA are presented, including their benchmark test results.

To validate algorithmic qualities of parallel SOMA three real-world engineering assignments were subject to optimisation – combustion engine optimisation, relay node placement in energy-constrained networks and aerodynamic wing optimisation.

ABSTRAKT

Dizertační práce se zabývá myšlenkou paralelizace evolučních algoritmů s primárním cílem vyvinout paralelní verzi algoritmu SOMA (SamoOrganizující se Migrační Algoritmus).

V první části práce je shrnuta historie a současný stav na poli distribuovaných výpočtů a paralelních evolučních algoritmů, což zahrnuje klasifikaci paralelních genetických algoritmů, popis paralelizačních schémat a přehled vybraných úspěšných aplikací paralelní optimalizace.

V následující části je popsána plně škálovatelná platforma pro paralelní/distribuované výpočty a několik možných paralelizačních modelů. Tato platforma představuje univerzální, velmi efektivní a snadno konfigurovatelný framework pro distribuované úlohy.

Následuje analýza existujících implementací paralelní Diferenciální evoluce a popis možností paralelizace algoritmu SOMA. Návrh čtyř různých paralelizačních schémat algoritmu je detailně popsán, včetně výsledků výkonnostních testů.

Práci uzavírá popis optimalizace praktických inženýrských problémů. K ověření optimalizačního výkonu paralelní verze algoritmu SOMA byly zvoleny tři úlohy: optimalizace nastavení spalovacího motoru, optimalizace umístění směrovacích stanic v bezdrátových sítích a aerodynamická optimalizace geometrie křídla letadla.

1 AIMS OF WORK

This work is aimed to parallel optimisation evolutionary techniques, especially to the very recent and powerful one – the Self-Organizing Migrating Algorithm (SOMA). The goal is parallel implementation of SOMA, its benchmarking and analysis of parallel SOMA running in environment of networked workstations.

The main issues considered and problems to be solved here can be summarized into the following points:

- to summarise the current state on the field of distributed computations and parallel evolutionary algorithms;
- to develop a platform for parallel computations. This platform will harness the processor-power of ordinary office computers mainly situated in the freely-accessible classrooms across the university campus and utilise their free CPU-time for parallel optimisations;
- to implement parallel SOMA in the master-slave and fine-grained configurations. This includes benchmarking tests and performance comparisons of all implemented parallelisation strategies..
- to employ the developed parallel SOMA for real-world time-demanding optimisation problems to validate its algorithmic qualities and high optimisation performance.

2 INTRODUCTION

Evolutionary algorithms (EA) based on principles of natural selection belong to very efficient methods of global optimization. They are successfully used in many engineering applications. In general, EAs are able to find a feasible solution of an optimization problem in a reasonable time. However, when using them for complex problems, the time required for finding a suitable solution might be unacceptably long. Therefore, many studies were published and lots of experiments were undertaken in effort to speed-up evolutionary algorithms. In run of the time, parallelisation became one of the ways of improving the computational performance of evolutionary algorithms.

Even though its youth, the Self-Organizing Migrating Algorithm (SOMA) (Zelinka, 2002; Zelinka, 2004) already proved that it can be regarded as a very powerful optimisation technique, an algorithm which can easily beat most of its predecessors on vast majority of optimised problems. Recently, the family of evolutionary algorithms welcomed a new member – SOMA – and now it is time to show its outstanding performance to the world.

There exist many successful parallel implementations of various genetic and evolutionary algorithms. Starting with Ant Colony Optimisation, Simulated Annealing, including Genetic Algorithm, Differential Evolution (Storn & Price, 1995) or many others, we can trace that all of them have already been parallelised. Until lately, SOMA remained the only algorithm without its parallel version. This work extends the SOMA's field of activity also on computational demanding optimisation problems, enabling it to walk in the immense world of challenging tasks. Sky is not a limit anymore.

The following literary exploration provides a brief view on the actual state in the field of parallel evolutionary algorithms. It describes parallel genetic algorithms, their classification, history of their development as well as the importance and influence of communication topology to the run of a GA, including some examples.

In the subsequent chapter we present a universal and high-performance computation framework, especially developed for optimisation using the Self-Organising Migrating Algorithm. Besides detailed description of its capabilities and performance there are several examples of parallel application that can guide you in your own implementation of parallel applications.

Thereafter, parallelisation schemes of Differential Evolution (DE) and SOMA are analysed and implementation of both algorithms is throughly described. Four different parallelisation strategies of SOMA are presented and compared, including their benchmark test results.

In the last chapter three very successful applications of optimisation using the parallel SOMA algorithm are demonstrated. While the first part describes an assignment where settings of a modern four-cylinder internal combustion engine were a subject to optimisation, the second presents a task in which positions of relay nodes in wireless network had to be optimised. The third and perhaps most impressive attachment of SOMA is the aerodynamic optimisation of wing geometry accomplished in cooperation with the Evektor company, a Czech leading civil aircraft manufacturer, for airplanes being already produced or coming to production very soon.

The main contribution of this work lies in the analysis, implementation and empirical validation of parallel SOMA evolutionary algorithm, which is primarily described in the "Parallel Evolutionary Algorithms – DE & SOMA" chapter.

3 STATE OF THE ART

3.1 Parallel genetic algorithm

Genetic algorithms (GA) are stochastic search algorithms based on principles of natural crossover and selection. They search for the optimal solution using manipulation with a population of individuals representing possible solutions. Single individuals in the population are evaluated by fitness – a value characterising quality of the particular solution. The best individuals are chosen for further population reproduction, which improves quality of the entire population.

The ground of the genetic algorithms theory is based on the theory of evolution formulated by Charles Darwin. Bad traits are eliminated from the population because they appear in individuals which do not survive the process of selection. The good traits survive and they are mixed by recombination (crossover, mating) to form better individuals. Mutation also exists in genetic algorithms, but is considered as a secondary operator. Its function is to maintain diversity among individuals, so the genetic algorithm can continue to explore the space of possible solutions.

Often, individuals are composed of binary strings of a fixed length L and thus GAs explore a search space formed by 2^L points. Initially, the population consists of individuals chosen randomly, unless there is a heuristic to generate good solutions for the domain. In the latter case, there is still part of the population generated randomly to ensure some diversity in the population.

The size of the population is an important parameter because it influences whether the GA can find a good solution in feasible time (Goldberg, Deb & Clark, 1992; Goldberg, 1994; Harik et al., 1997). If the population is too small, it can be difficult for the algorithm to identify good solutions. On the other hand, if the population is too large, the GA will waste computational resources processing unnecessary individuals. This balance between the quality of solution and the time the simple GA needs to find it also exists for parallel Gas; it will be explained how it affects their design later.

Each individual in the population is evaluated by its fitness. In general, fitness is a measure that depends on how well an individual solves a problem. In particular, GAs are often used as optimizers, and fitness of an individual is a return value of an objective function in a point represented by a binary string. The selection process uses this value to identify individuals that will reproduce and mate to form next generation.

Genetic algorithms adopted two operators from the natural genetics - crossover and mutation. Crossover is the primary exploration mechanism in GAs. This operator takes two randomly selected individuals from those already selected to form the next generation and exchanges random

substrings between them. As an example of crossover, consider the following individuals:

I1 = 1111 | 1111
I2 = 0000 | 0000

In a binary string of length L=8 there are L-1=7 possible points of crossover. In the example, the position of crossover is chosen as 4 and indicated by the symbol “|”. After exchanging the genetic information, the newly created individuals are:

I'1 = 1111 | 0000
I'2 = 0000 | 1111

The crossover operation can take many forms. The example above used 1-point recombination, but it is possible to use 2-point, n-point, or uniform crossover. More points of crossover result in a more exploratory search, but it can also cause extinction of individuals representing good solutions.

Mutation is usually considered as a secondary search operator. Its function is to restore diversity that might be lost from the repeated application of selection and crossover. This operator alters some random values within the binary string. The following example sketches mutation in a string of 8 bits on the 4th position:

I = 11111111
I' = 11101111

Likewise in the nature, the probability of applying mutation is very low in GAs, while the crossover frequency is usually very high.

There are several ways how to stop a genetic algorithm. One method is to stop after predetermined number of generations or function evaluations. Other is to halt the algorithm when the average quality of the population does not improve after some number of generations. Another often used alternative is to stop the GA when all individuals are identical.

3.2 Classification of Parallel Genetic Algorithms

The basic idea behind most parallel programs is to divide a task into chunks and to solve the chunks simultaneously using multiple processors. This divide-and-conquer approach can be applied to GAs in many ways and literature contains inexhaustible number of examples of successful parallel implementations. Some parallelisation methods use a single population, while others divide the population into several relatively isolated subpopulations. Some methods can exploit massively parallel computer architectures, while others better suit to computers with fewer but more powerful CPUs.

According to (Cantú-Paz, 2000), there are four main types of parallel GAs:

1. global single-population master-slave GA,
2. single-population fine-grained GA,
3. multiple-population coarse-grained GA,
4. hybrid GA.

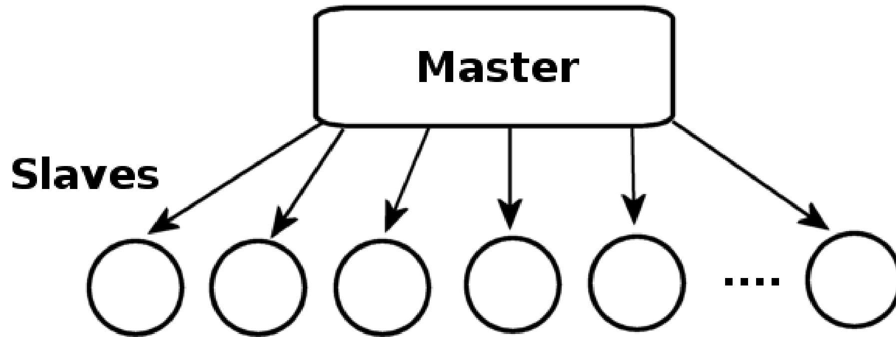


Figure 3-1*: An example of master-slave parallel GA. The master stores the population, executes the selection, crossover and mutation process and distributes single individuals to slaves. The slaves only evaluate the fitness of the individuals.*

In the master-slave GA there is a single population (just as in simple GAs), but the evaluation of fitness is distributed among several processors (see Figure 3-1). Since in this type of parallel GA the crossover and mutation handle the entire population, it is also known as global parallel genetic algorithm.

Fine-grained parallel GAs are suited for massively parallel computers and consist of one spatially-structured population. Selection and mating are restricted to a small neighbourhood, neighbourhood overlaps permit some interaction among all the individuals (see Figure 3-2). The ideal case is to have one individual available for every CPU.

The multiple-population parallel GAs are more sophisticated as they consist of several subpopulations which exchange individuals occasionally (see Figure 3-3). This exchange of individuals is called migration and is controlled in many different ways. Multiple-population parallel GAs are very popular. But they are also the class of GAs which is very difficult to understand because the effects of migration are not fully understood. Multiple-population parallel GAs have slightly different behaviour comparing to simple (serial) GAs.

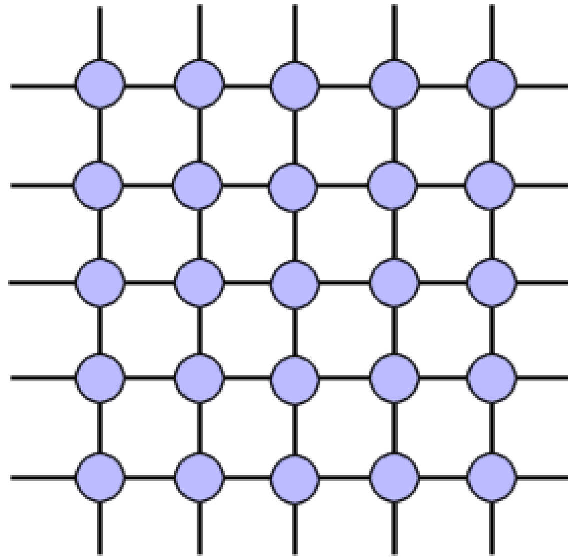


Figure 3-2*: An example of fine-grained parallel GA. This class of parallel GAs has one spatially-structured population and can be efficiently implemented on massively parallel computers.*

Multiple-population parallel GAs are known under different names. Sometimes, they are called as “distributed GAs”, because they are very often implemented on distributed memory computers (MIMD). In the case the ratio of computations and communication is very high, they are occasionally called “coarse-grained GAs”. Other example of parallelisation is the island model, when the subpopulations are relatively isolated from all others and they exchange individuals only rarely.

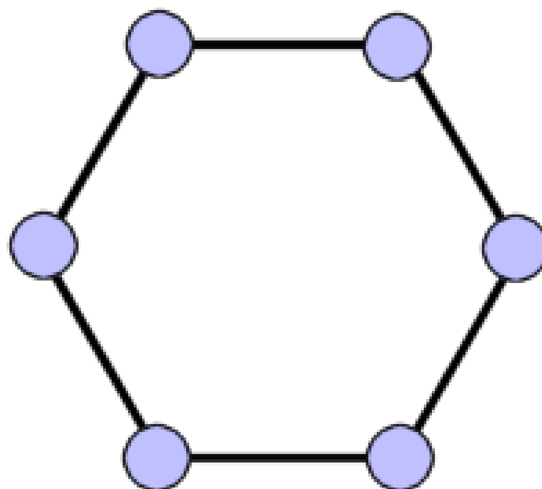


Figure 3-3*: An example of multiple-population parallel GA. Every process is a standalone GA which sometimes exchanges an individual with its neighbours.*

As emphasised in (Cantú-Paz & Mejía-Olivera, 1994) , the master-slave parallelisation method does not affect the behaviour of the algorithm, while the others change the way the GA works. The parallel version of master-slave GA takes into account the entire population where every individual can mate with all other ones, the other methods consider only a subset of individuals and limit the possibility of crossover only to one subpopulation.

The multiple-population coarse-grained parallel GAs combine multiple populations with the master-slave or fine-grained method. This class is called hierarchical parallel GAs. At higher level they are multiple-population algorithms with single-population parallel GAs (either master-slave or fine-grained) at the lower level. A hierarchical parallel GA can combine the advantages of its components and might offer a better performance than any of them alone.

3.3 History of Parallel Genetic Algorithms

As early as in 1976, Bethke described a global parallel implementation of a classic GA and GA with a generation gap (Bethke, 1976). He conducted an efficiency analysis and identified some bottlenecks that limit the parallel GA efficiency. Five years later, (Grefenstette, 1981) proposed four types of GA. The first three were variations of the master-slave scheme and the fourth one was a multiple-population GA. In 1985, Grosso presented an idea of realisation of a serial simulation of parallel GA run (Grosso, 1985). Its first implementation is described in the work by (Pettey, Leuze & Grefenstette, 1987a). They divided the population to several subpopulations, every element of their computation system had assigned one group of individuals that were processed separately.

In the same year Cohoon, Hedge, Martin and Richards adopted the theory of punctuated equilibria from the natural systems and applied it on artificial algorithms (Cohoon et al., 1987). In 1989, Belding became the first man who implemented GA on hypercube parallel computer (Belding, 1995). (Manderick & Spiessens, 1989) created the term of the island model parallel GA. Very important theoretical questions were raised by (Starkweather, Whitley & Mathias, 1991) about comparison of solution-quality between parallel and serial GA. They claimed that relatively isolated subpopulations converge to different solutions and that migration and crossover combine only partial solutions. A complete summary of the advances of the research in parallel genetic algorithms till 2000 could be found in (Cantú-Paz, 2000; Crainic & Toulouse, 1997; Cung et al., 2001).

The number of papers increased incredibly with the advances of computer sciences in the last decade (Alba & Troya, 2001; Alba & Troya, 2002; Cantú-Paz, 2000; Rivera, 2001). One of the very fruitful studies is the dissertation of Eric Cantú-Paz (Cantú-Paz, 2000). The author describes many new techniques of rational design of fast and accurate parallel genetic algorithms. Many researchers

follow his guidelines to decide about the topology configuration and number and size of subpopulations. He emphasises the importance of accurate design of population size, impracticability of isolated subpopulations, improvements in quality and efficiency by migration, advantage of fully connected topologies, effects of topology and optimal allocation of computing resources.

(Sefrioui & Périaux, 2000) proposed Hierarchical Genetic Algorithms with multi-layered hierarchical topology and multiple models for optimisation problems. (Rivera, 2001) reviewed strategies of parallel GAs including many implementations and results from their tests. Further, he discussed important issues regarding the scalability of parallel genetic algorithms.

(Alba & Troya, 2001) created a common framework for studying parallel GAs. The authors analysed the importance of synchronism in the migration step of various parallel distributed GAs. The synchronisation problem might influence the search time and also the speed-up. This study extended significantly the existing knowledge about structured-population GAs and demonstrated linear and even super-linear speed-up when run in a cluster of workstations (COW).

In (Alba & Troya, 2002), the authors compared the properties of steady-state, generational and cellular genetic algorithms and extended the idea to consider a distributed model consisting of the ring of the GA islands. Time complexity, selection pressure, processing rate, efficiency in finding the optimum, speed-ups and scalability were the analysed features. In addition, they briefly discussed how the migration policy affects the search.

(Xio & Amstrong, 2003) proposed a new model of parallel evolutionary algorithms (EAs) called Specialised Island Model (SIM). The model was derived from the island model, in which an EA is divided into several subEAs that exchange individuals among themselves. In SIM, each subEA is responsible for optimising its local subset of objective functions. Seven different scenarios with different number of subEAs and properties of various topologies are compared in this paper.

(Gagné, Parizeau & Dubreuil, 2003) compare the classic master-slave and the island models, which they implemented for Beowulf and a network of heterogeneous workstations (NOW). They identified the key features of a good computing system for evolutionary computation: transparency, robustness and adaptivity. As hard network failures caused many problems, they adjusted and extended the master-slave model in order to overcome them.

3.4 Master-Slave Parallel GA

Genetic algorithms in configuration master-slave use only a single population of individuals and the evaluation of individuals (and possibly also application of other genetic operators) are done

in parallel. As in the serial GAs, each individual may compete and mate with any other. Global parallel GAs are usually implemented as hierarchical master-slave programs, where the master stores the population and the slaves evaluate the fitness.

The most common operation that is parallelised is the fitness evaluation because fitness of an individual is independent on the rest of the population. Moreover, there is no need for communication during this operation. The evaluation of individuals is parallelised by assigning a single or a group of individuals to each available processor. After processing of assigned individuals, they are sent back to the master computer.

If the algorithm stops and waits to receive evaluated individuals from all slaves before proceeding into the next generation, then the algorithm is called synchronous. A synchronous master-slave parallel GA has exactly the same properties as a simple serial GA, where speed is the only difference. Although most of the parallel GA implementations is synchronous, there are also asynchronous implementations of some GAs where the algorithm does not stop to wait for any slow processors. However, they work in a different way than the simple serial GAs.

The global parallelisation model does not assume anything about the underlying computer architecture and can be effectively used on computers with shared or distributed memory. On shared memory multiprocessor systems is the population stored in shared memory and each of the processing units can access the assigned individual directly and write back its fitness directly without any conflicts.

On distributed memory systems, the population is stored on one computer – master. This master unit is responsible for assigning individuals (or group of individuals) to its subordinated computers – slaves. Slaves evaluate the fitness of received individuals and return them back to master, where other genetic operators are applied and new generation is created. The number of individuals assigned to a single processor may be constant, but in some cases (e.g. when exploiting computers in freely accessible classrooms where the utilization of CPUs is variable) it may be necessary to balance the computational load among the processors by using some dynamic task scheduling algorithm.

(Fogarty & Huang, 1991) attempted to develop a set of rules for a load balancing application. They used a network of transputers which CPUs were especially designed for parallel computations. Each of the transputers was able to communicate with any other either directly or by retransmitting messages. Because the message passing was a significant time consuming operation, they attempted to minimize this performance losses by interconnecting the transputers in different topologies. They

concluded that there is no significant difference in parallel computations performance caused by different network configurations. Although they obtained reasonable speed-ups, they identified the fast growing communication overhead as an impediment for further improvements in speed.

In (Abramson & Abela 1992), the authors implemented a parallel GA on 16-CPU shared memory computer to search for efficient timetables for schools. They reported only limited speed-ups and blamed some serial parts of code. Later, on a different machine (Fujitsu AP10000 with 128 CPUs) and with modified code, they reached a significant speed-ups for up to 16 processors, bud speed-ups degraded significantly as more processors were added, mainly due to the increase in communication.

Other implementation of a global parallel GA by (Hauser & Männer, 1994) gave other possible explanation - inadequate task scheduling. They claimed that it is necessary to consider all possible parameters influencing the overall performance when designing a parallel GA. It is a mistake to focus only on a single criterion.

Other aspect of GAs that can be parallelised is the application of genetic operators. Crossover and mutation can be parallelised using the same idea of partitioning the population and distributing the work among multiple computers. Nevertheless, this operations are computationally so undemanding that it is very likely that the time required for transferring individuals back and forth would neutralise any performance gains.

In conclusion, parallel master-slave algorithms are very easy to implement and it can be very efficient method of parallelisation when fitness evaluation is very time demanding while the communication costs are low. Besides, this method does not change the GA's behaviour, so we can apply all the theory already developed for serial genetic algorithms directly.

3.5 Multiple-Population Parallel GA

This class of parallel GAs works with several relatively large subpopulations and belongs to the most popular parallel methods. For that reason, there are many papers describing innumerable aspects and details of their implementation.

Probably the first systematic analysis of multiple-population parallel GA was Grosso's dissertation (Grosso, 1985). His objective was to simulate interaction of several parallel subcomponents of an evolving population. Grosso's population was divided into five subpopulations. Each subpopulation exchanged individuals with all others with a fixed migration rates. Grosso concluded that the improvement of average population fitness was the faster the smaller was the subpopulation. This confirms a long-held principle of population genetics – favourable traits

(e.g. traits ensuring higher probability of survival) spread faster when the population is small than in the case when the population is large. He also observed that when the subpopulations were isolated, the rapid rise in fitness stopped at a lower fitness value comparing to a large population. With a low migration frequency the subpopulations behaved independently and explored different regions of the searched space. The migrants did not have significant influence on the receiving subpopulation and quality of the global solution was then very similar to the case when the subpopulations were isolated. However, by reasonable migration rate, the divided population found solutions comparable to those found by a single population. These observations indicate that there is a critical migration rate below which the performance of algorithm is obstructed by the isolation of subpopulations, and above which the partitioned population finds solutions of the same quality as the GA with a single monolithic population.

(Pettey, Leuze & Grefenstette, 1987b) proposed a similar parallel GA. A copy of an individual is sent to the neighbouring subpopulations after every generation loop. The purpose of this constant communication was to ensure a good mixing of individuals. Like Grosso, the authors of this paper observed that parallel GAs with such a high level of communication found solution of the same quality like the serial GA with a single population. This observations prompted some questions like: (1) What is the level of communication necessary to make a parallel GA behave like a single-population serial GA? (2) What are the costs of this communication? And (3) are the communication costs low enough to make this a viable alternative for the design of parallel GAs? Some of them are not solved till now.

It is interesting that such important observations were made so long ago. (Tanese, 1987) proposed a parallel GA with subpopulations interconnected in the 4-D hypercube topology. In Tanese's algorithm, migration occurred at fixed intervals between processors along one dimension of the hypercube. Migrants were selected randomly from a group of the best in the source subpopulation and replaced the worst individuals in the receiving subpopulation. The most significant finding was that the increasing migration rate degrades the performance of parallel GA. After other numerous simulations she discovered that GA with n subpopulations and m individuals without any communication can reach the same results as the serial GA with n x m individuals. However, average quality of the final population is much lower.

A following paper by (Belding, 1995) confirms Tanesse's findings. The individuals were sent to randomly selected nodes connected in a hypercube instead to they neighbours in his experiment. Author claims, that in most of the cases the global optimum was found more often when migration was used than in the completely isolated cases.

3.6 Coarse-Grained parallel GA

In the early 90', when the potential and future of genetic algorithms was indisputable, the researchers started to concentrate on problems of speed-ups of parallel GAs and to understand better the way they work. Around this time, first theoretical studies began to appear and the theoretical research attempted to identify the favourable parameters. Parallel GAs started to be tested by large and very difficult test functions. One of the first contributions was the publication from (Mühlenbein, Schomisch & Born, 1991). They described a parallel GA which could find a global optimum of several functions which are now used as benchmarks for optimisation algorithms. In that time, there were many publications describing various successful implementations of parallel GAs, but theoretical studies were very rare.

Very important question is if, when and under what conditions is the parallel GA able to find better solution than the classical serial GA. (Starkweather, Whitley & Mathias, 1991) came with two observations regarding this questions. The first one was that relatively separated populations are likely to converge to different solutions and only partial solutions are shared and combined by migration. Authors speculated, whether the parallel GA with low migration rate is able to find better solutions than the one with high migration rate. Conclusion of the second observation was that if the recombination of partial solutions results in individuals with lower fitness, then the serial GA might have an advantage.

In the majority of parallel GA with multiple populations the migration is synchronous, which means that it is performed in predetermined constant intervals. However, migrations may also be asynchronous, controlled by some events. One such an example of this behaviour can be the Grosso's dissertation (Grosso, 1985), where he experimented with a delayed migration scheme. In this work the migration was enabled until the population was close to converge.

(Braun, 1990) used the same idea and presented very similar algorithm where migration occurred after the subpopulations fully converged. The purpose of the consequent communication (migration of individuals) was to restore the diversity in populations and thereby to prevent premature convergence into a low-quality solution. Afterwards, a number of authors studied the same strategy. (Cantú-Paz & Goldberg, 1997) presented theoretical models predicting quality of solutions when a fully connected topology was used.

At this point it might be interesting to raise a question: When is the right time to migrate? It seems that if migration occurs too early during the run, the quality of solution represented by migrating individual might be low and time-expensive communication resources would be wasted. On the

other hand when the individual is passed too late, a valuable information is shared to other subpopulations and thus the search process will be slowed down.

A different approach to migration was developed by (Marín, Trelles-Salazar & Hernández, 1994). They proposed a centralised scheme in which slave processors execute genetic operations on their local populations and send their partial results to a master process periodically. Then, the master process chooses the fittest individual found so far and distributes it to all slaves. Their experimental results proved the possibility of linear performance growth up to six processors. The authors claimed that this principle can be scaled up to larger number of processors because the communication is infrequent.

3.7 Fine-Grained parallel GA

Fine-grained parallel GA has only one population, but it has a spatial structure that limits the interaction among individuals. An individual can only compete and mate with its neighbours, but since the neighbourhoods overlap, a good solution can spread in the population. (Sarma & De Jong, 1996) analysed the influence of the size and the shape of the neighbourhood on the selection mechanism and found that the ratio of the radius of the neighbourhood to the size of the whole grid is a critical parameter which determines the speed of convergence of the whole population.

In this case, the individuals are allocated to a two-dimensional grid because the processing elements are connected in this topology in many massively parallel computers. However, most of this systems has also a global router that can send messages to any processor in the network (at a higher cost, naturally) and thus different topologies can be studied.

3.8 Communication topologies

Communication topologies are closely connected with the area of parallel genetic algorithms. The topology is an important factor in the performance of parallel GAs because it determines how fast or how slow is a good solution disseminated across the whole population. If the topology has a dense connectivity, good solutions are spread to other subpopulations fastly and may quickly dominate them. On the other hand, if the topology is sparsely connected, solutions are spread slowelier and the subpopulations are be more isolated from each other, resulting in higher number of different solutions. Recombination of these partial solutions may potentially lead to a better global solution. The communication topology is also important because it is a major factor in the cost of migrations. A densely interconnected topology may ensure better interaction of individuals, however, the communication costs will be higher.

The general trend of multiple population parallel GA is to use static topologies that are specified

at the beginning of the run of the algorithm and remain unchanged. Most of the parallel GA implementations with static topologies use the native topology of the computer system available. Therefore the topologies of a hypercube, a ring or a tree are common.

Some empirical studies like (Cantú-Paz & Mejía-Olivera, 1994) showed that parallel GAs with dense communication topologies find the global solutions using fewer function evaluations than GAs with sparsely connected ones. This study includes tests performed on networks in topologies of 4-D hypercube, 4x4 toroidal mesh and unidirectional and bidirectional rings.

The other major choice is the dynamic topology. In this method, any of the processors is forced to communicate with any concrete partner(s), but instead of that migrants are sent to subpopulations that meet some criteria. The motivation behind dynamic topologies is to identify the subpopulations where migrants are likely to produce some effect. Typically, the criteria might be the population diversity (Munetomo, Takai & Sato, 1993) or a measure of genotypic distance between the two populations (Lin, Punch & Goodman, 1994).

3.9 Hierarchical parallel GA

Some researchers tried to combine two or more methods of GA parallelisation, producing hierarchical parallel GAs. Some of these new hybrid algorithms add a new degree of complexity to the already complicated world of parallel GAs.

When two methods of parallelising GAs are combined, they form a hierarchy. At the top level, most of the hybrid algorithms can be viewed as multiple-population algorithms. Some of them have fine-grained GA at the lower level (see Figure 3-4). Other type of algorithms are those presented in Figure 3-5, where each island is a separate master-slave GA. The third variant of hybrid GAs are those with multiple populations on both top and bottom levels (Figure 3-6). The main idea of this approach is to force mixing of individuals on the low level using a high migration rate and a dense topology, while a low migration rate with limited mixing of individuals is used at the high level.

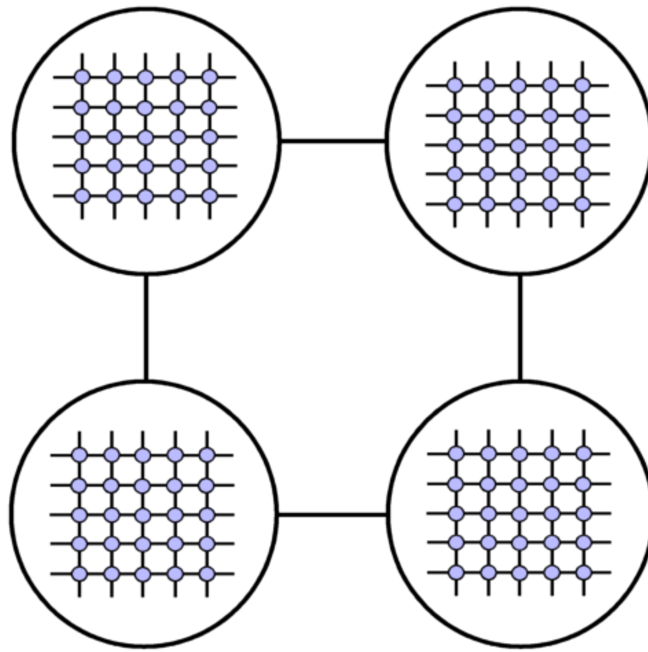


Figure 3-4*: An example of hierarchical parallel GA combining the island model on higher level and fine-grained GA on the lower level.*

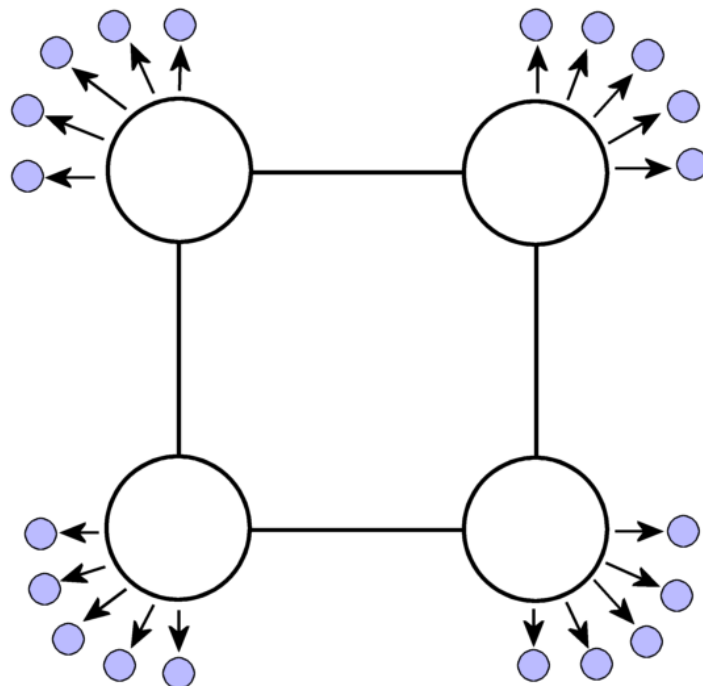


Figure 3-5: *Scheme of an hierarchical parallel GA. While the top level is considered as the island model, the lower level is represented by a master-slave GA.*

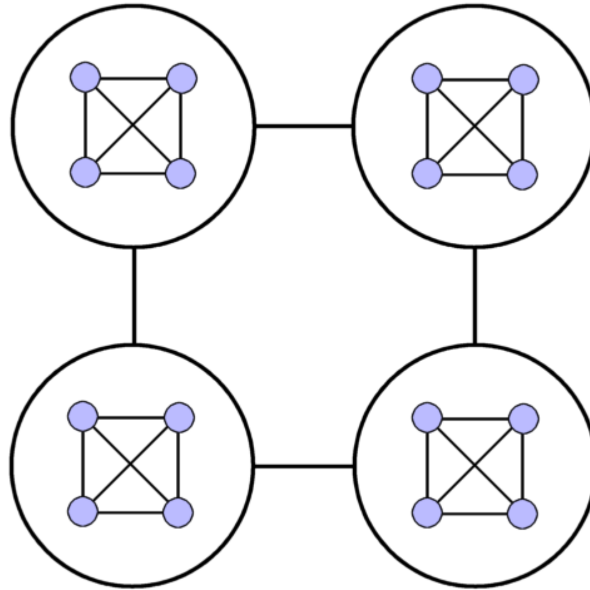


Figure 3-6: *An example of hybrid parallel GA with the island model on both levels. At the lower level, the migration rate is higher and the communication topology denser than at the upper level.*

3.10 Applications of parallel GAs

There is a countless number of occasions where to use genetic algorithms. Successful applications can be found from mathematics and the graph theory (numeric functions optimisation, scheduling problems, mission routing problems), in computer science (searching for weights of neural networks, optimisation of server load or database queries), in finance and economics (time series predictions, systems modelling, logistics problems) to technology and engineering tasks (VLSI circuits optimisation, optimisation of mechanic constructions or ultrasonic signals).

To the most impressive applications belongs the practical three-dimensional shape optimisation for aerodynamic design of a supersonic aircraft wing by (Oyama, Obayashi & Nakamura, 2000). The authors called the algorithm ARGA (Adaptive Range Genetic Algorithm), which had both binary and real value representations. Aerodynamic optimisations resulted in very enhanced wing design and proved the feasibility of the parallel genetic approach.

(Bevilacqua, Campanini & Lanconelli, 2001) investigated the improvement obtained by applying a parallel GA to a problem of parameter optimisation in the medical images analysis. The authors set a method for the detection of clustered microcalcifications in digital mammographs based on statistics and multiresolution analysis by means of wavelet transformation.

(Alba, Nebro & Troya, 2002) implemented distributed parallel GA in Java able to run on various

operating systems interconnected by different kinds of networks. This algorithm exploits the computation resources offered by modern LANs and the Internet. The authors analysed the way in which such heterogeneous systems affect the genetic search for two example problems.

In 2002, the new version of automatic distribution framework for evolutionary algorithms DREAM (Distributed Resource Evolutionary Algorithm Machine) was released. (Arenas et al., 2002) describes a virtual machine built from a large number of individual computers on the Internet. DREAM is a project on very high level of development. The highest level offers the possibility of graphical design of evolutionary algorithm. The lowest level of the framework is a P2P mobile agent system that could automatically distribute processes of the evolutionary algorithms.

4 PLATFORM FOR PARALLEL COMPUTATIONS

The goal of creating a platform for parallel computations was to provide a tool for time-demanding computations and simulations in the heterogeneous environment of Tomas Bata University's campus network. A classic computational cluster would require high investments in infrastructure and other necessary equipment. Computers would be used only for high performance computing tasks and in case when there are no tasks to process they would stay without any assignment.

To avoid very high initial costs when obtaining a high performance system there is an idea of using common office computers and connecting them to a virtual cluster. There is a high number of computers allocated across the entire campus, particularly in the library and freely-accessible classrooms, mainly used only for office-like applications where their CPUs idle for the most part of the computer's lifetime. These computers offer high performance on low or even no costs. Moreover, they are regularly maintained and upgraded. By employing this large group of computers we can obtain enough CPU time for our time demanding heavy computational tasks.

As an initial inspiration for this platform served the well-known seti@home project, which is based on similar principle. Its assignment is an analysis of an incredibly large amount of data acquired from the Arecibo radio-telescope observatory. Chunks of measured data are sorted and stored into databases and later distributed and analysed by computers offered by many people all around the world interested in this project.

Requirements on the cluster were formulated as follows:

1. General purpose platform

   For wide range of applications, effortlessly configurable.

2. Portable

   Server and also the terminal side must be able to run on arbitrary hardware and operating system.

3. Easy to use

   Development process of the cluster application must be as easy as possible, preferably without or with only a slight involvement of the user in the problems of parallel processing.

Used technologies:

- JAVA

  Applications developed at the Java platform are easily portable among various hardware and

operating systems. The only requirement is availability of Java Virtual Machine (JVM) for the target platform. Nowadays, there are virtual machines for every widely used operating system available on the Java's home site (java.sun.com).

- PHP

  Open source hypertext preprocessor widely used for web-based applications.

- Linux

  An outstanding operating system. Slackware 10.2 distribution was used for our server.

- MySQL/PostgreSQL

  Open source databases offering all needed functions and high performance meeting all our requirements.

4.1 Brief history of the platform

The aim of the former design was to develop an application interface (API) for distributed applications to enable to a commonly skilled user develop a parallel version of his algorithm in an easy way. On the server side, user created an instance of the ClusterServer class, which provided all necessary server services (information about the application progress, interface for creating, updating, and removing the task packages). The ClusterServer class was also a communication gateway for all terminals. On the terminal side, the ClusterTerminal class was used to provide communication channel to the server application. It provided also some other useful services as login, logout, getData and setData. The user did not have to be interested in the network communication problems and have any knowledge about the inner cluster functions.

This approach seemed to be really reasonable and advantageous that time, but only as long as the cluster did not need to be used for multiple applications. Moreover, this primordial version did not support automated class updating on the terminal side(s) and always, when there was a new build of the terminal application, the user had to go around all occupied computers and upload the modified classes manually. And this became to be very laborious task when we started to use more and more terminals.

For this purpose a really general-purpose platform for parallel computations was created. Although it became a little more complicated for the user, the new approach brought many useful improvements such as automated updating of terminal-application's classes on terminals performed by the UpdateManager, ability of transparent cluster management using the ClusterGuruGUI web

interface and also an ability to create virtual-topology computing networks.

During last few years the cluster computation platform underwent several major evolutionary steps and its last version represents very reliable and highly efficient framework for parallel applications.

4.2 General cluster description

The overall structure of the cluster software can be seen in Figure 4-1. After start of the ServerMng (i.e. start of the server), the ApplicationMng is run in a new thread. ApplicationMng starts or stops the user applications (ServerApplication) according to the project's activity flag in the database. ServerApplications run also in separate threads to be able to work concurrently. After initialization they create task-description/data packages (taskParts).

Start of the terminal (TerminalMng class) triggers the log-in and update process. First of all, the list of local classes is sent to the server. In case they are out-of-date, terminal receives new, actualized classes. The update mechanism updates both the user's and the platform's classes. Then, after initialization, a taskPart is requested from the server. Subsequently, a corresponding user's terminal application is launched and the data from the taskPart are passed as its parameter . Its return value is again a taskPart package, which is sent as a result to the server.

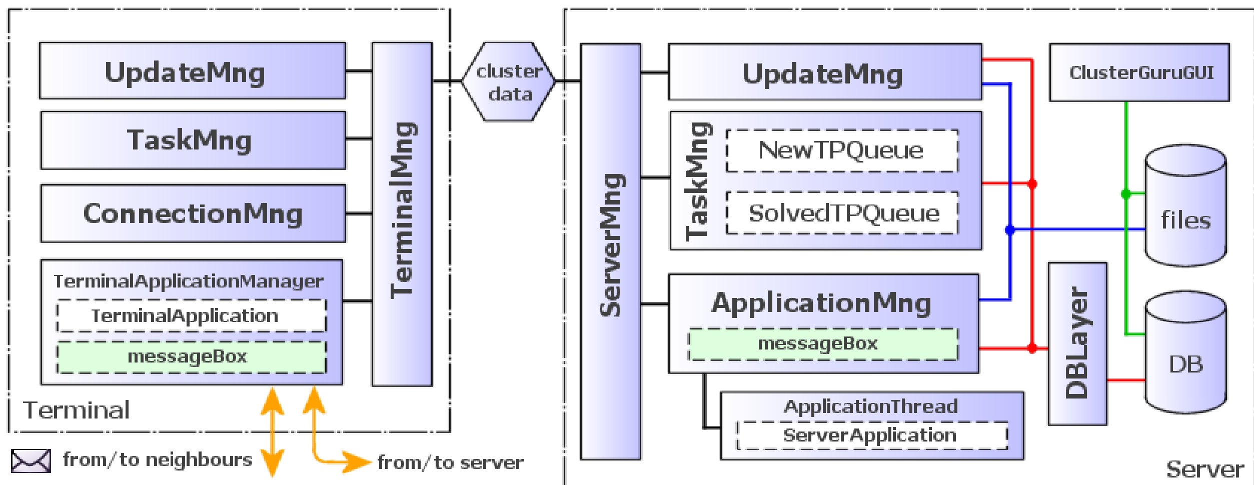


***Figure 4-1**: Cluster structure including interconnection of its inner modules*

TaskParts on the server side are distributed according to the flag indicating their status. After a taskPart is picked up by a terminal, its status is changed from `WAITING` (=to be processed) to `BUSY` (=being processed). In case the terminal does not return the taskPart during a certain period of time (the default taskPart's timeout can be set in the server configuration file, explicitly can be modified for every single taskPart when inserting it into database), the state of this package is changed to `WAITING` again. If a taskPart is returned after this period is over, it is stored to the

database. If any other terminal has already been assigned this task, the processing on such a terminal is interrupted and terminal receives a new job (if available).

Cluster applications can operate in two modes. In the seti@home-like mode the ServerApplication prepares numerous chunks of data to be analysed or many packages describing a partial assignment for the terminal. We call this packages *taskParts*. These taskParts are stored in the database (they might be optionally saved in separate files on the file system) and picked up by terminals. After being analysed/solved, they are sent back and saved into database. Later, when all taskParts are processed, the server application picks them up and assembles all partial results together. Terminals can not see any others and do not communicate to each other.

In the grid mode, terminals create a virtual-topology network. Initialization of application is done in the same way as in the first mode – terminals obtain an assignment from the server. However, they do not return immediate partial solution back, they cooperate together until the task is finished. In this mode, additional code is employed to establish a virtual network and to provide additional communication ability for terminals. This feature is described in chapter concerning the terminal's ConnectionMng class.

4.3 Database structure

All cluster-controlling flags and data (optionally) are stored in the '**cluster**' database. The following SQL snippets ambiguously define four tables.

The table **projects** describes the cluster application (the user's program). In the column *cpu_time* the used terminal's CPU time is aggregated and after the project has finished, it is possible to calculate the speed-up ratio as R = cpu_time / (stop_time – start_time).

```
CREATE TABLE `projects` (
  `id` text NOT NULL,
  `name` text NOT NULL,
  `description` text,
  `status` text NOT NULL,
  `start_time` bigint(11) default '0',
  `stop_time` bigint(11) default '0',
  `cpu_time` bigint(11) default '0',
  PRIMARY KEY  (`id`(128))
) ENGINE=MyISAM;
```

In the table **task_parts**, information with a task description and necessary data are stored . The column *status* may contain only the string `WAITING`, `BUSY` or `DONE`. If the time of processing a taskPart exceeds the `task_part_busy_timeout` value, its status is set back to `WAITING`.

```
CREATE TABLE `task_parts` (
  `project_id` text NOT NULL,
```

```
  `status` text NOT NULL,
  `terminal_address` text NOT NULL,
  `start_time` bigint(11) NOT NULL default '0',
  `data` longblob NOT NULL,
  `task_part_id` text NOT NULL,
  `temp_id` text NOT NULL,
  `timeout` bigint(11) NOT NULL default '0',
  PRIMARY KEY  (`task_part_id`(128))
) ENGINE=MyISAM;
```

Table **terminals** contains a list of physical terminals (computers). Each of the participating terminals has its own log, where the number of taskParts processed (*tasks_completed*) and the total CPU time used for computations (*cpu_time*) are recorded. The column *status* can only contain strings `CONNECTED` and `DISCONNECTED`. Terminals do not have to respond on a fixed communication port (this feature was added due to restrictions on some classrooms), they can automatically detect a free/opened port and adapt to local situation. The *address* and *port* columns are to keep the information how to reach a specific terminal (this is not important only for the server but also for other terminals when running an application in grid mode).

```
CREATE TABLE `terminals` (
  `location` text NOT NULL,
  `configuration` text NOT NULL,
  `status` text NOT NULL,
  `tasks_completed` bigint(11) default '0',
  `address` text NOT NULL,
  `logged_in_last_time` bigint(11) NOT NULL default '0',
  `id` text NOT NULL,
  `cpu_time` bigint(11) unsigned default '0',
  `port` bigint(11) unsigned NOT NULL default '0'
) ENGINE=MyISAM
```

Table **files** represents a list of classes and all other files that belong to each of the cluster projects. Files with the flag *destination* set to `TERMINAL` or `SHARED` are uploaded to terminals and automatically updated. `SERVER` files are intended to be launched only on the server side. Server and terminal application managers launch only classes with the flag *runnable* set to `YES`, all other classes are considered to be only supporting classes for user's applications.

```
CREATE TABLE `files` (
  `name` text NOT NULL,
  `file_name` text NOT NULL,
  `project_id` text NOT NULL,
  `type` varchar(5) NOT NULL default '',
  `runnable` varchar(4) NOT NULL default 'NO',
  `destination` varchar(8) NOT NULL default ''
) ENGINE=MyISAM;
```

4.4 Structure and functions of the platform

A detailed description of each of the cluster parts is given in this chapter. There are three packages – for the server side, for the terminal side and also a package of classes that are shared by both sides. The web-based cluster management interface is described in the next chapter.

4.4.1 Package cluster.server

This chapter describes in detail the functionality of the server side - process of logging in and out, automated updating of classes and the communication between server and terminal (see Figure 4-2).

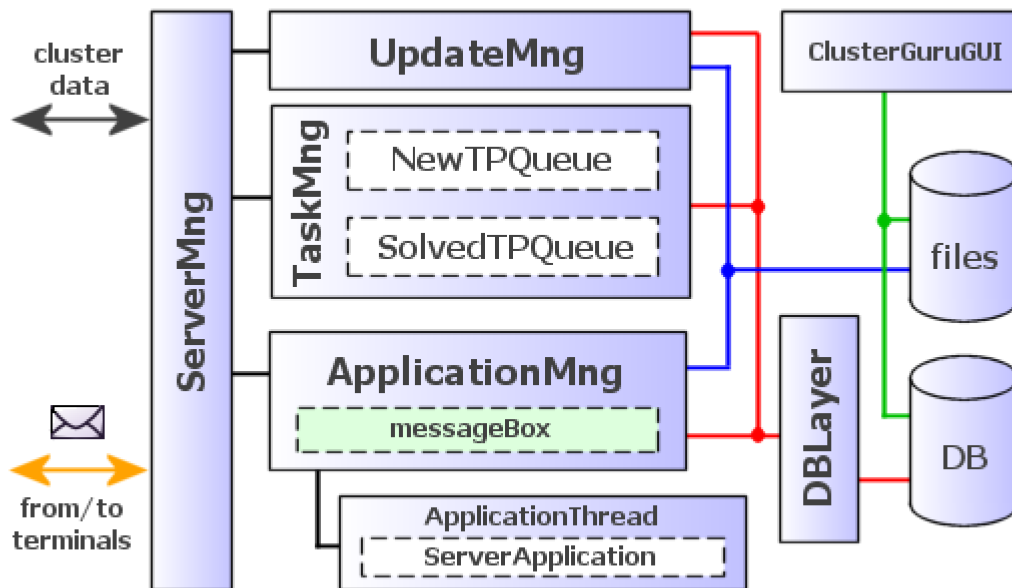


***Figure 4-2**: Schema of server structure*

ServerMng

Server manager is the main class of the server, which creates its own RMI registry and registers itself as an RMI service. Furthermore, it initializes a database connection pool that all other server modules will use. It offers the ServerMngI interface, through which all communication between server and terminals is routed to all other server modules.

```
public interface ServerMngI extends Remote {
   boolean login(TerminalDescriptor td) throws RemoteException;
   void logout(String terminalAddress) throws RemoteException;
   ClusterData getClusterData(ClusterData data) throws RemoteException;
   TerminalDescriptor[] getListOfConnectedTerminals() throws RemoteException;
   void passMessage(TaskPartBean tpb) throws RemoteException;
} ////:~
```

Vast majority of the messages are handled by the getClusterData() method. Three different events can be processed here:

- request for update → handled by the UpdateMng,

- request for an assignment (taskPart) → if there is any available, TaskMng returns data; if not, terminal is instructed what to do (typically to hibernate for a while and ask again later),
- incoming processed taskPart → passed to TaskMng.

ServerMng provides also some additional useful methods. First of them is login(), which registers terminal into database (if it is still not there, it will be added), sets its status to `CONNECTED` and records the time of last registration. It returns `true` if at least one parallel project is up and running. Method logout() was primarily created to maintain the list of online and offline terminals up-to-date. The passMessage() method is used when an application is running in the grid mode. It was implemented for direct access to the server-application's message box from all terminals.

UpdateMng

Update manager contains only one public method – update(). A list of classes owned by terminal is passed as a parameter. This list is then compared to the server's list of classes and files (the assumption is, that classes on server are always up-to-date) and those which terminal does not have or has older versions of them are returned. The return value is an array of the ClassDescriptor type. This data class contains a time stamp of last modification, name and size of the file, class name and also a binary copy of transferred class. The server's list is refreshed in the `FILE_LIST_RELOAD_INTERVAL` directive specified in the configuration file.

TaskMng

This module handles requests for taskParts and saves processed and new taskParts into a database. In order to balance high load caused by many terminals requesting and returning taskParts, two caches were implemented – the first one for taskParts waiting to be processed (NewTPQueue) and the second one for solved taskParts which are to be stored into database (SolvedTPQueue). Properties of both buffers can be modified using `QUEUE_LENGTH` and `QUEUE_CHECK_INTERVAL` parameters.

In the getTaskPart() method, all incoming requests for assignment are processed. If there is at least one project running (any project with status = `ENABLED`) and there are also any taskParts in the NewTPQueue, the return value is a TaskPartBean (TPB) containing besides the projectID and taskPartID the name of the class to be launched on terminal and data which are passed to the executed class as a parameter. Furthermore, the TaskPartBean carries also other useful information like cpuTime (processing time of the TPB, it is set when sending TPB back to server), taskPart timeout (taskPart's expiration time) and IP address of the terminal that processed the TPB.

The saveSolvedTaskPart() method funnels a processed taskPart to the SolvedTPQueue. This queue

is emptied by saving all contained TPB into database either periodically or when it is full (depends on which event comes first).

ServerApplicationMng

Server application manager is a separate thread, which checks the projects list in the database and compares it with the list of already running applications in a time period determined by the `projects_list_check_interval` parameter. If a change in the flag *status* is detected, a relevant action is performed:

- Transition from `DISABLED` to `ENABLED` → new ServerApplicationThread is created and in this thread a class with parameters *type*=`SERVER_APPLICATION` and *runnable* = `YES` in the '**project**' table is launched. The technique of starting the class will be explained later.
- Transition from `ENABLED` to `DISABLED` → according to the *project_id* the reference of relevant application thread is found and a termination signal is sent to it. The reference is then removed from the list. Nevertheless, the real halt of the thread depends on the ServerApplication and the user (programmer) has to take this signal into account and terminate his application (the exact procedure will be described later).
- Transition from `ENABLED` to `FINISHED` → the thread terminates itself, only the reference from the list of threads is removed.

Besides the above mentioned, the ServerApplicationMng provides useful supporting services for ServerApplication (the user's server application) via the following methods:

```
public boolean areAllTaskPartsSolved(String projectId) {....},
```

returns true if all taskParts have already been solved and the ServerApplication can pick them up.

```
public void setProjectFinishedStatus(String projectId) { ....}
```

This method is used just after all taskParts were reassembled to a complete result and the ServerApplication is going to terminate itself (finish). Internally, this changes the *status* flag (see the table '**projects**' above) from `RUNNING` to `SUSPENDING`, which tells to the ServerApplicationMng (SAM) to remove the current thread from the inner list of running applications. After that, SAM changes the status flag to the value `FINISHED`.

The following method

```
public void addNewTaskPart(String projectId, String taskPartId, long timeout, Object data) {....}
```

servers the ServerApplication to add new taskParts to a project. *TaskPartId* is better to be defined

by user (although it may bring some uncomfortableness) than automatically generated as it is more transparent way to maintain user's own database of created objects. Furthermore, here can be the value of taskPart's *timeout* specified. Then, the taskPart itself is passed as the *data* parameter.

```
public void removeTaskPart(String taskPartId) {....}
```

Using this method the user can remove an arbitraty taskPart form the project's list. Remember, that removing a taskPart which is currently processed from the database does not stop its processing on the terminal side. You can also remove all taskParts belonging to a project using the following method:

```
public void removeAllTaskParts(String projectId) {....}.
```

```
public Object getTaskPart(String taskPartId) {....}
public Object[] getAllTaskParts(String projectId) {....}
```

The methods mentioned above are used for withdrawing taskParts from the database. These methods do not remove the taskParts from DB, they must be removed explicitly by the user.

There are also methods for server-to-terminal(s) communication. They can be used in the grid mode and their definitions are as follows:

```
public TerminalDescriptor[] getListOfConnectedTerminals()
public synchronized boolean sendMessageTo(TerminalDescriptor td, String projectID, Object message)
public synchronized void sendMessageTo(TerminalDescriptor[] td, String projectID, Object message)
public Object[] getMessages(String projectID)
```

The first method is designed to obtain a list of currently connected terminals. Because the terminals connect and disconnect during run of the project, it is very recommended to re-read this list more often than only once after start of the project. The other two methods are intended to send a message to one or to a selected list of terminals (to a message box identified by the *projectID*). The getMessages() method picks up all messages delivered to the local message box – messages for the ServerApplication.

Process of launching a ServerApplication consists of the following steps:

- An instance of a class with flags **destination** = `SERVER` and **runnable** = `YES` is created. This is done using the ServerApplicationI (*sai*) interface and the Class.forName() call.
- New ServerApplicationThread (*sat*) is started and instance of ServerApplicationI (*sai*) is passed as *sat*'s contructor parameter. Then, *sat* calls the *sai*.go() mehtod and the server application is launched.
- When a thread with server application is running, its reference is saved

to a ServerApplicationMng's list of active threads (ServerApplicationThreadsTable). This table is also the place, where the ServerApplicationMng can obtain an instance of ServerApplicationI to pass it a termination message.

To reduce the network load, ServerApplicationMng can optionally compress the taskParts using the ZIP algorithm. Activation of this feature is defined by the `use_compression` directive in the configuration file. However, it is good to remember that in some cases zipping numerous and/or large taskParts can considerably increase the server load (at least during some time after the ServerApplication was started) and thus eliminate any benefits gained from network load reduction.

ServerApplicationThread

ServerApplicationThread stands for a thread which encapsulates the ServerApplication. It calls the go() method defined by the ServerApplicationI interface and thereby launches the ServerApplication. It is the only point of communication contact between the server application and ServerApplicationMng and also the only way how to stop the server application.

ProjectDescriptor

ProjectDescriptor is a data class in which information about project is stored. It is used by ServerApplicationMng in ServerApplicationThreadsTable to observe any changes in states of defined projects.

Database layer

The database layer represents an intermediate layer between the database and server modules. It maintains a pool of database connections and provides all necessary services for ServerMng (logging in and out, keeping up a list of terminals), UpdateMng (maintaining and updating a list of files that have been changed), TaskMng (handling with taskParts) and ApplicationMng (all services needed when launching, terminating and working with the user's server application). This unit creates some kind of translator between the world of SQL and world of objects. Furthermore, it is a primary point where the change in status of a project is detected (e.g. a transition between `RUNNING` to `TERMINATING`) and in which an appropriate event is started. It also observes the state of taskParts stored in the database (a 'zombie watch'). In the case when solving/analysing/processing of a taskPart exceeds its time-out (which in most cases means that the terminal on which the task had been assigned has crashed or has been restarted), this taskPart is redeployed to the other terminal requesting a task.

4.4.2 Package cluster.terminal

Implementation of the terminal side (see Figure 4-3) is thoroughly explained in this section.

It includes description of the log-in and log-out process, the procedure of updating local files and classes (including terminal itself). Also the method of requesting taskPars from server is depicted. The last part of this chapter covers the virtual-grid feature, its potential, advantages and characteristics.

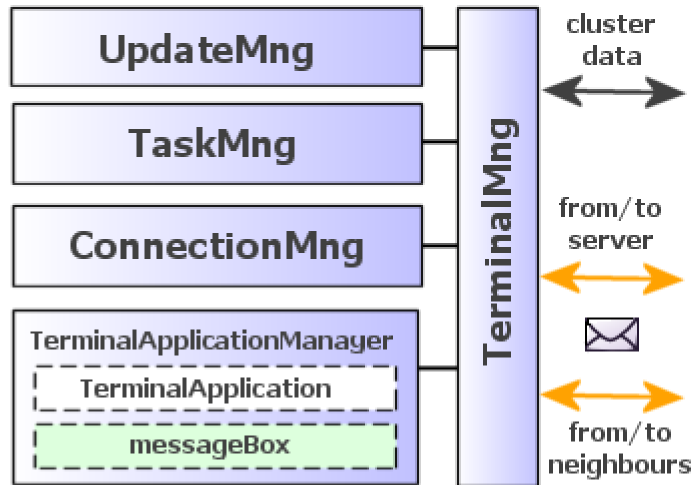


Figure 4-3*: Schema of terminal structure*

TerminalMng

Terminal is a standalone application typically running as a system service (on NT) or a detached process (on Linux) on many computers across the university's campus network. Terminal manager is the main class of the terminal application.

When the TerminalMng is launched, connection to the server as an instance of remote object is created. If the server is down or not responding for some other reason, terminal becomes suspended for a time period given by the `HIBERNATE_TIME` directive. If the connection is successfully established and terminal is logged-in, the first update procedure is started. This process does not involve only terminal application files, it also includes update of the terminal itself. Subsequently, the TaskMng, UpdateMng and ConnectionMng are initialized and the terminal is waiting to receive an assignment.

TerminalMng also provides the TerminalMngI interface for other terminals to be able to connect and communicate to each other. It offers the following methods, their functionality is described in the "ConnectionMng" section.

```
int ping() throws RemoteException;
void passMessage(TaskPartBean tpb) throws RemoteException;
boolean connect(TerminalDescriptor td) throws RemoteException;
boolean disconnect(TerminalDescriptor td) throws RemoteException;
```

Classes used by the user's terminal application are stored in a directory where the `TERMINAL_CLASSES_STORAGE_DIRECTORY` directive from the terminal's configuration file points. Also the `$CLASSPATH` system variable should have an entry pointing to this directory. If it has not, the TerminalMng tries to set it up by itself, but it may not succeed. For the critical case when the classes are not loaded by the system class loader, there is a specially designed class utils.CL (ClassLoader) which serves as a custom class loader and also as a last stage when all other attempts fail.

UpdateMng

The update manager looks after local classes and files to keep them up-to-date. It includes maintaining a list of versions in the file *.file.list* stored in the `TERMINAL_CLASSES_STORAGE_DIRECTORY`. This file contains a serialized array of BinaryFile classes describing each file stored on the terminal computer.

When UpdateMng is initialized, the list of local files is compared and, if necessary, modified to reflect the actual state. Then, an update request is periodically sent to the server in order to get up-to-date classes and files. The period is given by `FILES_AUTO_UPDATE_INTERVAL` directive, where 5 minutes is a recommended interval. Note that this interval strongly depends on size of classes and data files you wish to transfer to terminals. The user should consider the network load the update process may cause.

Server response to the update request is again an BinaryFile array or *null* as a sign that there is nothing to be updated.

TaskMng

Terminal's task manager communicates directly with the server's task manager. It is a module which in `TASK_MNG_CHECK_INTERVAL` requests taskParts from the server. When a processing of a taskPart is started, TaskMng requests immediately other task to minimize the delay between the moments when one task is finished and sent back to server and a new one is obtained; the subsequent task is started right after the previous one has finished and before it is submitted to the server.

TerminalApplicationMng

Terminal application manager is a simplified equivalent to the ServerApplicationMng. Only one user's terminal application can be run at the same time. TerminalApplicationMng provides a few services for the terminal application and you can see them below.

```
public Object[] getMessages() {....}
```

```
public boolean sendMessageToNeighbours(Object message) {....}
public boolean sendMessageToServer(Object message) {....}
```

These methods have sense only in the grid mode and have the following meaning: the first one returns all messages which have been received for the running user's application. Messages are assorted according to the *projectID* which comes with the taskPart and is supplied by the TaskMng. The other methods are to send messages either to the server or to the terminal's neighbours. How are the neighbours defined is described in the following section. As you can see, the type of a message is the Object class, which means that you can send any kind of a message. The only requirement is that the class must implement the Serializable interface.

The runnable class representing the terminal application must obligatorily implement the TerminalApplicationI interface. For an illustration how to implement a custom terminal application see the examples below.

ConnectionMng

Connection manager is a brand new terminal module which has been added recently. It is designed to enable terminals to communicate among each other and to create a kind of a virtual network. Its basic functionality lies in performing the process of logging in and out and maintaining the server connection.

The advanced functionality and the so-called *cluster grid mode* was primarily developed in order to parallelize the SOMA evolutionary algorithm in a way to support the evolutionary process (see the chapter about parallel SOMA). For progress of SOMA it is better to propagate a leading individual from one to other computation nodes (terminals) slower than faster. In other words, it is better to share more (albeit local) leaders than to have only a single global leader. This causes higher diversity in the population and thus may lead to better results.

For that reason, the basic configuration of the virtual network is a mesh where each terminal has four neighbours. There are no edges in the network and you can imagine it, for example, as to be covering a surface of a sphere. The information about a leader spreads step-by-step trough neighbouring terminals and in the case the leader is of high quality, it may gradually become a global leader.

Establishing of this virtual mesh is the advanced mission of ConnectionMng and it proceeds in the following steps:

1. A list of all currently connected terminals is obtained from the server.
2. 'Ping' of all terminals from the list is performed. This is to sequence terminals according to

their ping time and also to get the information about the number of free slots. The term *free slots* indicates the number of neighbouring terminals that the pinged terminal is missing.

3. Four terminals with best pings and at least one free slot are asked to establish a connection.
4. If the free slots are not taken by other terminals in the mean time between the ping and connection request, a connection is established. Otherwise a subsequent terminal from the list is requested to establish a connection.

As terminal computers may be restarted, turned off or even crash, all connections to the neighbouring terminals are periodically checked. If a connection to a neighbour fails, the procedure described above is repeated to find the missing connection(s).

What is more, even when all terminals in the network are fully connected, the network is periodically reconfigured in order to find neighbours with the lowest value of ping time for every terminal (to find the best location in the virtual mesh). Whenever a terminal with a better ping and a free slot is found, connection to this terminal replaces the connection to the worst terminal already connected.

Each terminal running in the grid mode establishes its own RMI registry to enable other terminals (and also the server) to communicate with them. Due to various restrictions (e.g. firewalls) in the freely-accessible classrooms and IT-labs the RMI registry port is not a fixed value, the terminal can dynamically adapt to the local situation. Because of this, information about a terminal port is associated with the remote terminal IP address and is included in the list of terminals obtained from the server.

4.4.3 Package cluster.shared

The cluster.shared package contains classes shared by the server and also by the terminal side.

BinaryFile

BinaryFile represents a class or a general file that is transported during the update process. It contains all necessary information about a file. It is used by the update manager either as a file descriptor without any data inside (local file list) or as a full-bodied data file including the binary image of the file (for transportation of the file).

ClusterData

The class ClusterData (see Figure 4-4) stands for a general wrapper for all other data objects sent among server and terminals. There are four possible types of this object: `UPDATE`, `DATA_REQUEST`, `DATA_RESULT` and `TASK_DATA`.

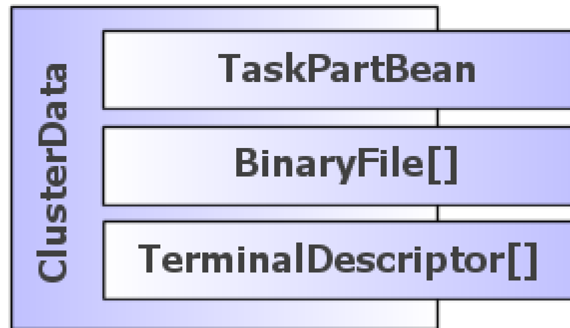


Figure 4-4: *Data classes that are transported within ClusterData*

TaskPartBean

This is a transportation data class for the taskPart – the assignment for a terminal. It contains the following items:

- projectID – to identify the project,
- taskPartID – to identify the taskPart,
- className – a user terminal application's class that processes this task,
- cpuTime – time spent with processing this task (used for terminal and project statistics),
- timeout – after this time is the taskPart status in the database changed from `BUSY` to `WAITING`. Then, it might be assigned to other terminal. The taskPart has to be solved by this time,
- terminalAddress – address of the terminal which processed this taskPart,
- data – an argument/return value of the terminal application main class.

The data transported within the TaskPartBean may be optionally compressed.

TerminalDescriptor

TerminalDescriptor class supplies useful information about a terminal – its IP address, port of its RMI registry where it accepts connections and messages from neighbouring terminals or the server, terminal location and configuration, ping time, and also the instance of a remote terminal as an object (TerminalMngI) when a terminal is connected. It is used by the terminal ConnectionMng.

ClusterException

ClusterException stands for an exception class that represent a wrapper for all exceptions that may

be raised in the cluster (except RemoteException). Its purpose is to reduce the number of exceptions that might be passed to the user application and filter cluster inner exceptions.

FixedPortRMISocketFactory

FixedPortRMISocketFactory overrides the original Java's RMISocketFactory in order to be able to set the RMI's responding port. In general, RMI service creates a port on which incoming connections are accepted. When a connection is accepted, the socket moves to other port, which is used for communication (i.e. data transfer). To be able to specify the responding port, it was necessary to modify the RMISocketFactory implementation.

MessageBox

MessageBox represents a postman that delivers, receives and sorts messages sent among terminals and server.

4.4.4 Package cluster.utils

ClassLoader (CL)

ClassLoader is a custom class loader used when starting a user's terminal application and the *runnable* class was not loaded by the Java class loader for some reason (the class is not in `CLASSPATH` or has been downloaded after last restart of the terminal).

MyProperties

MyProperties is class facilitating access to cluster configuration file. It periodically observes changes in configuration and modifies the cluster parameters.

MyUtils

The MyUtils class embodies several useful methods that can be also used by the user's application. They are:

- getUniqueId(String prefix) – helpful for generating IDs for taskParts,
- zipObject(Object o) and unzipObject(byte[] b) to compress/decompress data objects,
- loadTaskPartFromFile(String taskPartID) and saveTaskPartToFile(String taskPartID, Object data) – provide a direct access to taskParts when they are stored in files instead of the database and the user wants to bypass the standard taskPart handling mechanism. The location where the taskParts are stored can be changed by setting the `USE_DB_FOR_TASK_PARTS` directive in the cluster configuration file to 0 or 1.

4.5 *ClusterGuruGUI*

All tools needed for administration of cluster and user applications are accessible via a graphic user interface (GUI) on the web. This approach was chosen primarily for its easy accessibility for anyone from anywhere having just a web browser. Using this interface you can create, modify, launch or terminate your parallel projects. Furthermore, you can see the actual state of running projects, the time the running projects need to be finished and also the total time spent on processing a particular project. Owing to this web management interface, users can comfortably access all results without having direct access to the server's console. The other advantage is that the cluster administration is accessible from various mobile devices like cell phones, PDAs and others.

ClusterGuruGUI 4.1

[:show all projects:] [:add project:] [:terminals list:]

Name	Description	ProjectID	status	TaskParts (W/Q/B/F)	StartTime	StopTime	Duration	CPU time	Ratio
Distributed SOMA		dsoma	FINISHED [:start:]	(0/0/0/0) **0% F**	20-04-2006 14:04	20-04-2006 22:25	8h 20min 40s	13d 23h 26min 48s	40.20
GRID-1	Grid test #1	GRID-1	FINISHED [:start:]	(0/0/0/0) **0% F**	04-05-2006 09:30	04-05-2006 11:05	1h 34min 40s	1d 6h 23min 16s	19.26
Mathematica Cluster	Grid version	mathcluster	TERMINATED [:start:]	(0/0/0/0) **0% F**	15-12-2005 08:00	-	-	-	-
Reboot	Reboots all terminals	reboot	TERMINATED [:start:]	(0/0/0/0) **0% F**	21-04-2005 09:37	21-04-2005 09:41	3min 51s	3min 0s	0.78
Test1	pop_Test1	id1	TERMINATED [:start:]	(0/0/0/0) **0% F**	13-01-1970 22:26	-	-	-	-
Test2	pop_Test2	id2	RUNNING [:start:]	(254/200/66/80) **13% F**	20-05-2006 11:36	(running)	(22min 49s)	11h 30min 1s	30.25
TimeCluster	Efficiency test	time	FINISHED [:start:]	(0/0/0/0) **0% F**	01-05-2006 11:12	01-05-2006 15:28	4h 15min 47s	2d 16h 35min 7s	15.15
TrafficCluster		traffic	FINISHED [:start:]	(0/0/0/0) **0% F**	13-04-2004 23:28	13-04-2004 23:29	1min 7s	-	-

TaskParts : (W/Q/B/F) = (WAITING/QUEUED/BUSY/FINISHED)

http://cluster.utb.cz © ibisek MMIII-MMVI

Figure 4-5: *Main page of cluster's administration interface. Here you can see state of a parallel project, start and stop time, CPU time used for processing this assignment and also the speed-up ratio. In addition to this, it is possible to modify the project properties, to start/terminate or create new parallel project.*

On the main page of the cluster administration centre, you can see a list of projects wrapping parallel applications. Here is displayed the project name, its brief description, projectID (which uniquely identifies the project's files and taskParts), status (including a button to start or stop the project), taskParts summary (indicating the number of taskParts that are waiting to be processed (the *W* flag), queued in the task manager's buffer (*Q*), being processed (*B*) and already finished (*F*)). In addition to this, there is also the time when the project has started and the in case the project has finished by its own, you can find the project's stop time as well. As duration of a project run can be calculated from these two values and as also we log and store the sum of used CPU time, we can get the speed-up ratio. This ratio can be understood as a number of CPUs participated on the parallel

project with 100% efficiency.

To create a new project, click on the [:add_project:] button. You can set up here the project name, its brief description and an ID which will be used as a unique identifier for project files (used by the update manager) and taskParts (handled by task manager).

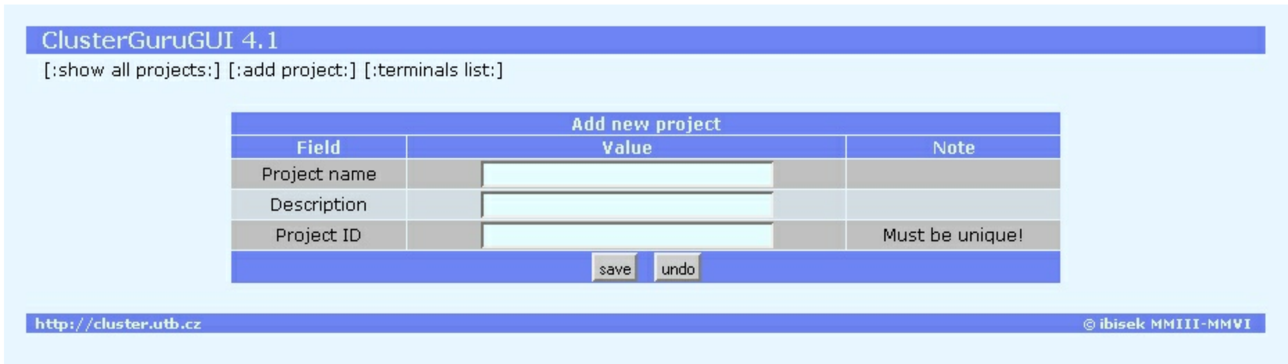


Figure 4-6: *Creating new project*

ClusterGuruGUI 4.1

[:show all projects:] [:add project:] [:terminals list:]

Project settings	
Field	**Value**
Project name	Distributed SOMA
Description	
ProjectID	dsoma
save undo	

Files						
Class name	**Filename**	**Type**	**Runnable**	**Edit**	**Destination**	**Remove**
SomaServer	SomaServer.class	CLASS	YES		SERVER	
	soma.server.conf	TEXT	NO		SERVER	
SomaTaskPart	SomaTaskPart.class	CLASS	NO		SHARED	
SomaTerminal	SomaTerminal.class	CLASS	YES		TERMINAL	
OptimizedFinctionI	OptimizedFunctionI.class	CLASS	NO		TERMINAL	
SchwefelFunction	SchwefelFunction.class	CLASS	NO		TERMINAL	
Specimen	Specimen.class	CLASS	NO		TERMINAL	
	soma.terminal.conf	TEXT	NO		TERMINAL	

Add a new file		
Field	**Value**	**Note**
File	Procházet...	
Filename	set	
Type	CLASS	For type CLASS you MUST set the ClassName value. For other types could stay the value ClassName blank.
ClassName		Server/Terminal will create instance by this name For type CLASS only.
Type	SERVER	TERMINAL and SHARED files are distributed to terminals
save clear		

[:remove project:]

http://cluster.utb.cz © ibisek MMIII-MMVI

Figure 4-7: *Project settings. Here you can add and remove classes/files to/from a project.*

By clicking on the project name on the ClusterGuruGUI's main page you access the project properties. Besides the project name and its description, project attributes can be modified here (see Figure 4-7).

Classes and other files belonging to the project are uploaded, maintained and removed here. There are five types of files - `CLASS`, `BLOB`, `TEXT`, `XML`, `PROP`. Only one class can be set as runnable on both the server and the terminal side. `TEXT`, `XML` and `PROP` files can be modified using a simple text editor. As a last attribute, the *destination* flag of a file must be defined. Files are transferred to connected terminals (*destination* set to values `TERMINAL` or `SHARED`) according to this parameter.

The [:terminals_list:] button redirects you to the terminals inventory. Here are all terminals that participate on parallel computations in the cluster. Their IP address, status (`CONNECTED` or `DISCONNECTED`), location of the terminals' computers, CPU type, number of completed tasks and used CPU time can be seen on this page.

ClusterGuruGUI 4.1

[:show all projects:] [:add project:] [:terminals list:]

Summary				
Terminals total	Terminals connected	Terminals disconnected	Tasks completed	CPU time
74	4	70	395929	315d 17h 48min 25s

[:logout all terminals:]

Terminals list								
Address	Port	Status	Location	HW_Cfg	Tasks completed	Logged in last time	Last logged in before	CPU time
195.178.89.153	4006	CONNECTED	ankh	P4/1800	339	02-05-2006 17:42	3h 6min 24s	23min 6s
195.178.89.198	4006	CONNECTED	beowulf	Celer800	358	02-05-2006 17:47	3h 1min 24s	26min 22s
195.178.95.208	4006	CONNECTED	U5-lib	Athlon XP 2200+	1025	02-05-2006 17:09	3h 39min 44s	17h 5min 1s
195.178.89.168	4006	CONNECTED	alypsia	P4/2400	1205	02-05-2006 17:59	2h 49min 44s	21min 1s
195.178.89.202	4006	DISCONNECTED	B204	kdovico	0	13-01-2005 20:41	473d 23h 7min 36s	0s
195.113.97.88	4006	DISCONNECTED	B205	Athlon2200	1034	02-05-2006 17:25	3h 23min 4s	1h 5min 21s
195.113.97.81	4006	DISCONNECTED	B205	Athlon2200	1232	02-05-2006 17:25	3h 23min 4s	1h 42min 13s
195.113.97.83	4006	DISCONNECTED	B205	Athlon2200	1234	02-05-2006 17:25	3h 23min 4s	1h 42min 19s
195.113.97.82	4006	DISCONNECTED	B205	Athlon2200	1238	02-05-2006 17:25	3h 23min 4s	1h 42min 42s
195.113.97.74	4006	DISCONNECTED	B204	kdovico	1279	02-05-2006 17:25	3h 23min 4s	1h 46min 44s
195.113.97.84	4006	DISCONNECTED	B205	Athlon2200	1470	02-05-2006 17:25	3h 23min 4s	1h 49min 46s
195.113.97.86	4006	DISCONNECTED	B205	Athlon2200	1485	02-05-2006 17:25	3h 23min 4s	1h 32min 48s
195.113.97.87	4006	DISCONNECTED	B205	Athlon2200	1486	02-05-2006 17:25	3h 23min 4s	1h 33min 16s
195.113.97.85	4006	DISCONNECTED	B205	Athlon2200	1489	02-05-2006 17:25	3h 23min 4s	1h 33min 7s
195.113.97.90	4006	DISCONNECTED	B205	Athlon2200	1543	02-05-2006 17:25	3h 23min 4s	1h 9min 55s

Figure 4-8: *Terminals list and their statistics*

4.6 Cluster configuration file

After ServerMng or TerminalMng is launched, the configuration file is processed and server/terminal modules are initialized using these values. While on the Linux platform the default configuration is stored in *c4.linux.conf,* the name of the configuration file is *c4.win32.conf* on Windows This chapter presents an example of typical cluster configuration file.

When creating a package to be distributed (installed) on terminal computers, it is recommended to remove the server settings section as it contains sensitive information. On terminals there is a risk that the configuration file might be misused by a potential intruder.

```
# c4.linux.conf

# ------------------------------------------------------------------------
# Server settings
# ------------------------------------------------------------------------

DB_URL = localhost
DB_PORT = 3306
DB_USER = cluster
DB_PASSWORD = ****
DB_NAME = cluster4

QUEUE_LENGTH = 200
QUEUE_CHECK_INTERVAL = 4000

# taskParts can be stored in DB or in files
USE_DB_FOR_TASK_PARTS = 1
# when user_db_for_task_parts = 0 it stores task parts to following directory. Put '/' as the last character.
TASK_PARTS_STORAGE_DIRECTORY = _task_parts/

# location where the user's application data is stored (both terminal and server) Put '/' as the last character.
SERVER_FILES_STORAGE_DIRECTORY = _server_files/

# interval of terminal's hibernation when receives a WAIT message:
GET_TASK_REPEAT_DELAY_FOR_TERMINAL_START = 1000
# with each additional number of terminals
GET_TASK_REPEAT_DELAY_FOR_NUM_TERMINALS = 2
# increase wait:
GET_TASK_REPEAT_DELAY_FOR_TERMINAL_INCREMENT = 1000

# project list check interval (for the server's ApplicationMng)
PROJECT_LIST_CHECK_INTERVAL = 5000

# Compress the taskParts with ZIP?
USE_COMPRESSION = 0

# UpdateMng:
FILE_LIST_RELOAD_INTERVAL = 10000

# ------------------------------------------------------------------------
# Shared settings
# ------------------------------------------------------------------------
```

```
DEBUG = 1
# LOGGING_LEVEL - see java.util.logging.Level
# values: OFF, SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST, ALL
LOGGING_LEVEL = ALL
LOG_FILE = cluster4.log

RMI_REGISTRY_URL = 195.178.89.168
RMI_REGISTRY_PORT_SERVER = 4004
RMI_SERVICE_NAME_SERVER = cluster_server

# Port on which RMI will handle request after accepting on RMI_REGISTRY_PORT:
RMI_FIXED_RESPONDING_PORT = 4005

# ------------------------------------------------------------------------
# Terminal settings
# ------------------------------------------------------------------------

# information describing terminal's configuration and location:
TERMINAL_HW_CONFIGURATION = P4/2400
TERMINAL_LOCATION = alypsia

# if there's nothing to do, wait this interval and ask again:
HIBERNATE_TIME = 10000

# directory, where terminal application's classes are stored.
# MUST BE IN CLASSPATH! Put '/' as the last character.
TERMINAL_FILES_STORAGE_DIRECTORY = _terminal_files/

# how often perform terminal files update (5min)
FILES_AUTO_UPDATE_INTERVAL = 300000

# how often to check if next new task part is downloaded to terminal.
TASK_MNG_CHECK_INTERVAL = 1000

SHOW_PROGRESSBAR = 1

# Terminal's default port and RMI service name (used in the grid mode only):
RMI_REGISTRY_PORT_TERMINAL = 4006
RMI_SERVICE_NAME_TERMINAL = cluster_terminal
# Number of neighbours in grid. Zero turns off the grid mode.
NUMBER_OF_NEIGHBOURS = 4
```

The configuration file can be located in the operating system temp directory (e.g. /tmp), in user home directory (Java *user.home* system property) or preferably in the Cluster distribution root directory.

4.7 How to write a cluster application

This chapter describes two sample applications to provide instructions for a potential user of the cluster.

4.7.1 Example #1 – Primes

In this section, a very simple parallel application searching for primes is presented. This example

finds primes within a given range. The assumption is, that the numbers are in the range of *double*.

Version for one processor could look like this snippet of code:

```
/**
 * @return true if the number given by argument is a prime.
 */
  public static boolean isPrime(double number)  {
          double maxDivisor = (double)Math.ceil(Math.sqrt(number));
          double divisor = 2;        // ( inverzni delitel ;)

          boolean isPrime=true;
          while(divisor < maxDivisor) {
                    if(Math.ceil(number/divisor)==(number/divisor)) {
                              isPrime=false;
                              break;
                             }
                    divisor++;
                   }
          return isPrime;
  } ////:~

/**
 * Validates numbers in given range whether they are primes or not.
 */
  public static void findPrimes(double fromNumber, double toNumber)  {
          for(double i=fromNumber ;i<=toNumber;i++) {
                    System.out.print("Validating number "+i);
                    if(isPrime(i))  {
                              System.out.println("Number "+i+" IS a prime.");
                             }
                   }
  } ////:~
```

How to parallelize this code? We can move the validation of each number on a separate processor. To reach this, for every single number the server part of the parallel application creates a task (taskPart) which will be analysed on the terminal side. The taskPart for this example might look like this data class:

```
//PrimesData.java

import java.io.*;
public class PrimesData implements Serializable {
          private double number;
          private boolean state;

          public void setNumber(double number) {
                   this.number=number;
          }
          public double getNumber() {
                   return number;
          }
          public void setState(boolean state) {
                   this.state = state;
          }
```

```
	public boolean isPrime() {
		return state;
	}
}///:~
```

Note that this class **must implement the Serializable interface**, because it is serialized for saving into the database and also for the network transfer.

The terminal application detects if the number is prime or not. Its implementation is very simple, the method isPrime() is the same as in the one-cpu version.

```
//PrimesTerminal.java
import cluster.shared.TerminalApplicationI;

public class PrimesTerminal implements TerminalApplicationI {

	public PrimesTerminal() {}

	public static boolean isPrime(double cislo)  {
		// the same code as in the one-cpu version
	}

	public Object go(Object parameters) {
	     PrimesData dta = (PrimesData)parameters;
	     dta.setState(isPrime(dta.getNumber()));
	     return (Object)dta;
	}
}///:~
```

As you can see, the PrimesTerminal class implements the TerminalApplicationI interface with the mandatory go() method that accepts Object in its argument and returns Object as well. Handling the Object within the go() method is fully up to the programmer. In this case, the argument is our PrimesData class defined above. The number stored inside this data class is checked for being or not being a prime number, the result is using the setState() method saved into the data class and PrimesData are returned to terminal's application manager. Then, the taskPart is sent to the server and another one is obtained to be processed.

Implementation of the server side of the primes-application follows:

```
// PrimesServer.java
import cluster.server.applicationmng.ServerApplicationI;
import cluster.server.applicationmng.ServerApplicationMng;

public class PrimesServer implements ServerApplicationI {

	private boolean go = true;
	private ServerApplicationMng sam;
	private final static String projectID = "primes";
	private final static long timeout = 30000; // 30 seconds
```

```
	public PrimesServer() {
		sam = ServerApplicationMng.getInstance();
	}

	private void createTaskParts(double fromNumber, double toNumber) {
		sam.removeAllTaskParts(projectID); // delete old task_parts
		 PrimesData pd = new PrimesData();
		// first we will create task parts:
		for(double i=fromNumber; i<=toNumber; i++) {
			pd.setNumber(i);
			// the analyzed number will be used as the taskPartID:
			String taskPartID = "task_"+String.valueOf(i);
			sam.addNewTaskPart(projectID, taskPartID, timeout, (Object)pd);
		}
		System.out.println(" All task_parts were created.");	// printed to the server's console
	}

	public void go() {
		int i;
		double fromNumber = Double.parseDouble("8888888888880000");
		double toNumber = Double.parseDouble("8888888888881000");

		createTaskParts(fromNumber, toNumber);

		while(go) {
			try { Thread.sleep(10000); } catch (InterruptedException ex) {}
			if(sam.areAllTaskPartsSolved(projectID)) go = false;
			System.out.println("PrimesServer : areAllTaskPartsSolved =
				"+sam.areAllTaskPartsSolved(projectID));
		} //while

		sam.setProjectFinishedStatus(projectID);

		Object[] obj = sam.getAllTaskParts(projectID);
		PrimesData[] pdat = new PrimesData[obj.length];
		try {
			for(i=0;i<obj.length;i++) {
				pdat[i]=(PrimesData)obj[i];
			}
		} catch(ClassCastException ex) {
			System.out.println("PrimesServer : *** ClassCastException ***");
			ex.printStackTrace();
		}

		System.out.println("\nThe following numbers are primes :");
		for(i=0;i<pdat.length;i++) {
			if(pdat[i].isPrime()) {
				System.out.println(" Number "+pdat[i].getNumber()+" is prime");
			}
		}
		System.out.println("PrimesServer : APPLICATION FINISHED");
	}

	public void stop() {
		go = false;
	}
} ////:~
```

The PrimesServer class implements the ServerApplicationI interface in which there are two mandatory methods – go() and stop(). The go() method is the main method of the user server application. The taskParts are created in the first step. Then the server application waits (in the while loop checking result of the sam.areAllTaskPartsSolved() method) until all taskParts are processed. Afterwards, as all taskParts are processed and (this) application is not going to create any other batch of tasksParts, the project status is set to `FINISHED`. Finally, the taskParts are picked up from the database and obtained results are printed into the server console (can be written to a file or processed in another way).

The stop() method is to tell the user's server application to stop when the flag status in the database is changed by the user via the ClusterGuruGUI. The programmer must take into consideration this change.

Instance of the server application manager is obtained in the PrimesServer's constructor.

The described program searching for prime numbers is just an example application. Most of the numbers are divisible by small numbers (divisors) which practically results in an immediate response from the terminal (taskPart is processed in a very short time). If there was a higher number of terminals processing such a short tasks and instantly requesting new assignments, the server load would be very high. Therefore it is advised to aggregate small tasks into bigger units and thus save the server and network resources.

4.7.2 Example #2 – TimeCluster

The original intention of creating this group of classes was to measure the cluster efficiency. This parallel application represents a simulation of a heavy computational task. The terminal application pretends a 10 seconds long computation (waits for 10 seconds). There are 600 taskParts created, 10 seconds each. This stands for a 6000 seconds long computation (=100 minutes). Dividing this value by the number of active terminals (terminals participating on the test), we get a theoretical time needed to process all taskParts. Then we compare this value to the real time in which the project was completed and get the cluster efficiency (see the subsequent Cluster performance chapter).

Let us have a look at the source code. Here is the shared data class:

```
// TimeData.java
public class TimeData {
        private long delay;
        public void setDelay(long delay) {
                this.delay=delay;
        }
        public long getDelay() {
                return delay;
```

```
        }
} ////:~
```

Source of a simple terminal application is:

```
//TimeTerminal.java
import cluster.terminal.TerminalApplicationI;
public class TimeTerminal implements TerminalApplicationI {
        public TimeTerminal() {}

        public Object go(Object data) {
            try {
               Thread.sleep(((TimeData)data).getDelay());
            } catch(InterruptedException ex) {}
        return data;
        }
} ///:~
```

And here is the corresponding server application:

```
//TimeServer.java
import java.util.Date;
import java.util.Random;
import cluster.server.applicationmng.ServerApplicationI;
import cluster.server.applicationmng.ServerApplicationMng;

public class TimeServer implements ServerApplicationI {
      private TimeData td;
      private Random rnd;
      private ServerApplicationMng sam;
      private boolean go = true;

      private final static String projectID = "time";
      private final static long timeout = 30000; // 30 seconds

      public TimeServer() {
            td = new TimeData();
            rnd = new Random();
            sam = ServerApplicationMng.getInstance();
      }

      /**
        * @param String prefix
        * @return String unique ID
        */
      public String getUniqueID(String prefix) {      // creates ID of time[ms] + rnd int
            String retVal = prefix+String.valueOf(new
            Date().getTime())+"_"+String.valueOf(Math.abs(rnd.nextLong()));
            return retVal;
       }

      private void createTaskParts(long delay, int numPacks) {
            sam.removeAllTaskParts(projectID);  // delete old task_parts

            for(int i=0 ;i<numPacks;i++) {
                  td.setDelay(delay);
                  String taskPartID = getUniqueID("time_task_");
```

```
                sam.addNewTaskPart(projectID, taskPartID, timeout, (Object)td);
            }
    }

    public void go() {
            long delay = 10000; // 1000 = 1s
             int numPacks = 600;
            System.out.println("TimeServer : createTaskParts started");
             createTaskParts(delay,numPacks);
             System.out.println("TimeServer : all task_parts were created");

             while(go) {
                try { Thread.sleep(1000); } catch (InterruptedException ex) {}
                if(sam.areAllTaskPartsSolved(projectID)) go=false;
                System.out.println("TimeServer : isFinished = "+sam.areAllTaskPartsSolved(projectID));
             }

            sam.setProjectFinishedStatus(projectID);
            sam.removeAllTaskParts(projectID);   // clean-up
            System.out.println("TimeServer : FINISHED");
        }

        public void stop() {
            go = false;
        }
} ///:~
```

4.8 Cluster performance

Every distributed application has certain efficiency. Loss of performance may originate on server, where the server side is not capable to handle terminals requests fast enough (for example due to slow database responses). Other factor decreasing cluster efficiency might be high load of the server computer, which results in slowing down the user server application that can not create new taskParts as fast as terminals request them and thus terminals have to wait. Performance might also be reduced by slow communication among computers caused by network components (long time delays when opening a socket, slow data transfers).

Besides the network throughput and the server (computer) performance, other two factors are brought by the cluster platform. The first one is the project-state check interval. The test is executed in a period given by the `PROJECTS_LIST_CHECK_INTERVAL` directive. It means that there is a latency between the start of the project using the web interface (status of the project in the **projects** table is changed to `ENABLED`) and the thread with user server side application is actually launched. Very similar factor is the interval of calling ServerApplicationManager.areAllTaskPartsSolved() method (returns true if all taskParts have already been processed; used in the user server side program where it waits to pick all taskParts up and to reassemble the partial solutions to a global one). If these two periods were too short it would result in higher server load, which may lead to some problems described in the previous paragraph

(recommended value is about or more that 500ms).

The goal of the tests was to determine the cluster efficiency. The experimental applications TimeCluster, PrimeCluster and DataCluster were tested in the library classroom where there are 41 Athlon XP 2200+ computers with 512MiB RAM on the 100BASE-TX network, running Windows. Server hardware configuration was Intel Celeron at 800 MHz, 768MiB RAM with Slackware Linux installed. Every simulation setpoint was performed 10 times and measured data represent its average value.

Presented data was measured during weekends when the classrooms are not available to students and the terminal computers run without any accidents. In workday-opening hours, the cluster efficiency might be lower because of students rebooting or even turning-off computers running the cluster terminal program.

4.8.1 TimeCluster

TimeCluster is a purely testing parallel application. There are 6000 taskParts created, computation time of each taskPart is 10 seconds (10 seconds long wait on the terminal side). This task in one-CPU configuration should last 6000 * 10 s = 1000 min ~ 16 hours 40 minutes. After splitting this task, for example, to 10 computers, the theoretical time to finish this parallel task is 100 minutes. The difference between the real and the theoretical time needed to process this task is the investigated cluster efficiency.

In Figure 4-9 and Table 4-1 one can see that with increasing number of terminals the cluster efficiency very slightly decreases. This might be caused by higher task request rate handled by the TaskMng. In addition to this, the task manager project check interval setting causes losses of several seconds deteriorating the overall performance. However, comparing to previous cluster versions, where the efficiency was around 96% (version 3), the cluster efficiency is now fairly higher. Results provided by the profiler shows, that most of the delays are caused by server and terminal settings.

TimeCluster	Delay [s]	10			
	Num taskPart	6000			
Number of computers	**1**	**10**	**20**	**30**	**40**
Time of processing [s]	6054	604	303	203	152
Ideal time of proc. [s]	6000	600	300	200	150
Delay [s]	54	4	3	3	2
Delay [%]	1	0,67	1,00	1,50	1,33
Efficiency [%]	99,11	99,34	99,01	98,52	98,68

Table 4-1: *TimeCluster – test results*

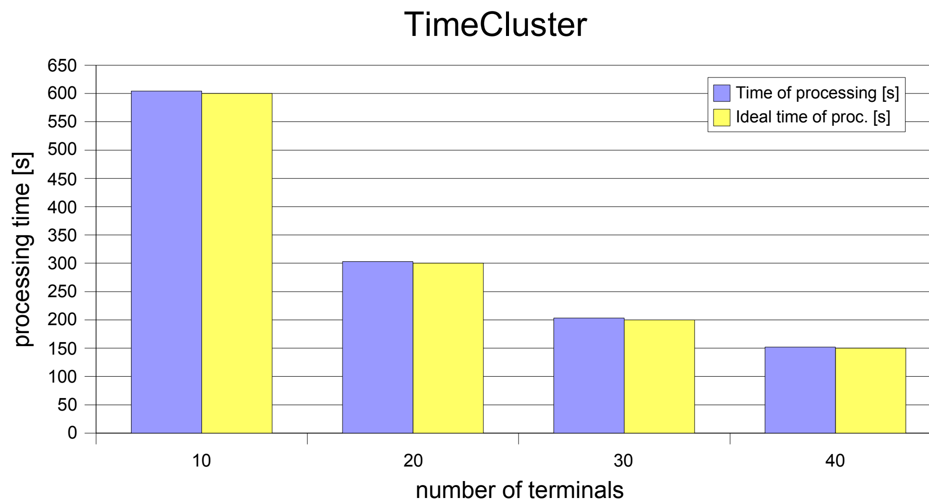


Figure 4-9: *TimeCluster – test results*

4.8.2 PrimeCluster

This example was primarily used as a test of server task manager's performance. Most of the tasks were returned to the server almost immediately, which caused some kind of a 'taskPart storm'. There were no significant time-delays detected. This set of experiments proved that the NewTaskPartQueue and SolvedTaskPartQueue modules together with the completely rewritten and improved DatabaseLayer stand no longer for a bottleneck in the cluster performance.

4.8.3 DataCluster

The DataCluster test was designed to determine the network throughput influence on the cluster efficiency. Messages of a size from 10kB to 2MB were tested with simulated data processing time on the terminal side. This resulted more in a network performance test than a test of the cluster platform. In general, we can claim that running a cluster on the 100BASE-TX network with applications we need at most (evolutionary algorithms) absolutely complies our requirements.

4.9 Chapter Summary

This chapter describes the first necessary step in the process of SOMA parallelisation. The cluster platform stands for a foundation stone on which the further work is built up. It represents a fully scalable, high-performance, universal and multiplatform framework for parallel/distributed applications. This versatile cluster platform was designed in order to utilise free CPU time of ordinary office computer that are present in relatively high numbers at all universities and many companies. As it can be seen from the test results, the platform demonstrated very decent performance and high reliability during all performed tests. With efficiency about 99% it is ready to be used anywhere where there is a need for high-performance computational tasks.

5 PARALLEL EVOLUTIONARY ALGORITHMS – DE & SOMA

5.1 Differential evolution

Differential Evolution (DE) is a modern optimisation method capable of handling nondifferentiable, nonlinear and multimodal objective functions. DE has been designed as a stochastic parallel direct search method that utilizes concepts borrowed from the broad class of evolutionary algorithms. The method typically requires few easily chosen control parameters. Many experimental results have shown that DE has good convergence properties and proves better performance than other well known evolutionary algorithms like Simulated annealing, Genetic algorithm or Ant colony optimisation. It is based on principles similar to genetic algorithms – it works with a population of individuals that mate in generation-steps. Kenneth Price's and Rainer Storn's work (Storn & Price, 1995) is one of the early applications involving the idea of Differential evolution.

5.1.1 Serial DE

The goal of DE is to cultivate the best possible individuals in terms of their cost function values. During run of the Differential evolution algorithm the following steps are performed:

1. **Specification of parameters** – parameters controlling the evolution process are determined. Related parameters are:

   - $F \in <0, 2>$ – the mutation constant,
   - $CR \in <0, 1>$ – crossover threshold,
   - NP – number of individuals in the population. The most used population size is $NP = 10*D$, where D denotes dimensionality of the optimized problem.

   Furthermore, a prototype individual (*specimen*) has to be defined – to classify the parameter-types (e.g. integer, double, string etc.) of the individual.

2. **Generation of initial population** is done by creating individuals according to the prototype vector (specimen). Every individual consists of a set of parameters for each dimension of the optimized problem and one extra value – the cost value describing fitness of the individual.

3. **Evolutionary loop**. Individuals are taken one by one and the following step is performed during every generation.

4. **Evolutionary step**. To the one active (source/target) selected individual other three from the population are randomly chosen. The first is subtracted from the second one which gives so called *differential vector*. It is multiplied by the mutation constant F that alters (mutates) it.

The product, *weighted differential vector*, is added to the third selected individual, which gives a *noise vector*. From the noise vector and the active individual, a trial vector is assembled – a random number in range <0, 1> is generated and confronted to the crossover threshold (*CR*) constant. If this number is lower than *CR* then a component (element in the vector) to the trial vector is taken from the noise vector, from the active vector otherwise. This is done for every pair of elements in the vectors. Further, the trial vector is evaluated and if its cost value is lower than the source vector's (individual's) cost value, the source individual is replaced by the newly spawned one.

The whole population is modified in order to replace old (worse) individuals with better ones. By returning to step #3, next individual is selected and the evolutionary loop repeats until all individuals are processed. This way a new generation of offsprings (individuals) is created.

5. **Stopping criterion test**. DE is terminated after specified number of generations (given by user). This algorithm does not have any other stopping parameter.

5.1.2 Parallel DE

As mentioned in the chapter about parallel genetic algorithms above, there are several possible strategies how to parallelise an evolutionary algorithm. Various parallel models have been proposed (Cantú-Paz, 2000; Adamidis, 1998):

- objective function evaluation level (master-slave model), where only the evaluation of the objective function is parallelised,
- population level (multi-population model, also called island or migration model),
- elements level (called as cellular, diffusion or neighbourhood model).

The cellular model leads to fine-grained parallelisation while the other two lead to coarse-grained parallelisation.

In (Tasoulis et al., 2004), the authors present a coarse-grained approach. They employ the Parallel Virtual Machine (PVM) framework, which de facto stands for a standard message passing interface. It is and integrated set of software tools and libraries that encapsulate a general-purpose, flexible, heterogeneous concurrent computing framework on interconnected computers of varied architectures. PVM is designed to link computing resources and to provide user a parallel platform for running his computational applications. It is capable of harnessing the combined resources of typically heterogeneous networked computing platform to deliver high levels of

performance and functionality. Its key concept is that it makes a collection of computers to appear as one large virtual machine, hence its name Parallel Virtual Machine (Geist et al., 1994).

DE, like other evolutionary algorithms, can be very easily parallelised due to the fact that each member of the population is evaluated individually. The only phase of the algorithm that requires communication among individuals is reproduction. This phase can also be parallelised for pairs or quartets (depends on modification of DE) of individuals. Authors remind that there are two strong models for parallelisation of evolutionary algorithms. The first employs fine-grained parallelism, in which each individual is assigned to a separate processor. However, this approach is problematic when the number of available processors is limited or when the computation of the fitness function requires information from the entire population.

The second model, that the authors of (Tasoulis et al., 2004) chose, maps a whole subpopulation to one CPU. Thus each subpopulation evolves independently toward a solution. This allows each subpopulation to develop its own solution independently without being influenced by a progress in other subpopulations. To promote information sharing, the best individual of each subpopulation is injected to other subpopulations according to a predefined topology. This operation is called *migration* (do not confuse with the term *migration* in SOMA). Authors propose a topology in which computing nodes form a ring, where individuals from each subpopulation are allowed to migrate to the next subpopulation of the ring. This concept reduces the migration among subpopulations and also the number of messages exchanged among processors. Migration of best individuals is controlled by the *migration constant* ϕ, $\phi \in <0, 1>$. At each iteration, an uniformly distributed random number in the interval $<0, 1>$ is chosen and compared with the migration constant. If the migration constant is larger, then the best individual of each subpopulation migrate and take the place of a randomly selected individual (different from the best one) in the next subpopulation on the ring. Otherwise no migration is performed.

A high level description of the algorithmic scheme follows:

At the master node:

1. Spawn N subpopulations; each one on a different processor.
2. For each generation:
3. Receive an individual from each subpopulation.
4. Determine for each individual if it is going to migrate. Decision on migration is based on the ϕ parameter.

5. Perform migration of chosen best individuals to the next subpopulation on the ring.
6. If the termination criterion for the objective function is met, send a stop signal to all subpopulations.

At each subpopulation:

1. For each generation:
2. Perform a DE step.
3. Send the best individual to the master node.
4. Receive a migrated individual if such one exists and assign it to a random individual.
5. If a termination signal is received, terminate execution of the loop.

Authors also suggest a modification based on ageing of individuals. This would prevent individuals from surviving indefinitely and should improve robustness of the algorithm.

In the experimental section, the authors of (Tasoulis et al., 2004) primarily studied impact of μ (mutation) and ρ (recombination) constants. Brief analysis of the influence of the migration constant ϕ on the performance of parallelised DE showed that selecting an appropriate migration constant has a significant impact on the performance of the parallelised algorithm. It appears that setting ϕ close to zero or one leads to a substantial decrease in the algorithm efficiency. A superior performance is typically obtained for intermediate values of ϕ. Further, the paper (Tasoulis et al., 2004) discuss the influence of the μ, ρ and ϕ parameters on the robustness of the algorithm. Unfortunately, any deeper investigation of dependency of parallelised DE performance on the migration parameter was not made. Presented values of speed-up were performed only for value of $\phi = 0.5$ and showed rather low efficiency of described parallel DE implementation.

(Zahaire & Petcu, 2003) investigated a parallel implementation of the Differential evolution also based on the multi-population model. Their reasons for choosing this approach were inspired by existence of the spatial structure in the natural populations and ability of this structure of preserving the population diversity through the migration process.

To avoid difficulties related with the parameters choice of parallelised DE mentioned in (Lampinen, 1999), the author propose his own adaptive modification of the DE algorithm, in which they claim the parameter settings should not be a problem.

The model presented in (Zahaire & Petcu, 2003) is based on dividing the population into *s*

subpopulations of the same size μ. On each subpopulation, an adaptive DE is executed for a fixed number τ of generations. Each DE corresponding to a subpopulation works with its own set of randomly initialized adaptive parameters. After each τ generations, a migration process (based on random connection topology) is started. More specifically, the migration strategy follows this principle: each individual from each subpopulation can be swapped (with a given *migration probability* p_m) with a randomly selected individual from a randomly selected subpopulation (including the subpopulation which contains the initial individual).

Due to the migration process, a subpopulation with a low diversity can be "revived" after the migration takes place. Hence the multi-population approach allows avoiding premature convergence situations in DE. Zahaire and Petcu demonstrated this by numerical results obtained for several benchmark test functions. They simulated a random topology between the processes of a parallel code on PC cluster with eight Pentium 4 1500 MHz with 256 MiB RAM interconnected via a Myrinet switch and optical fibre cables ensuring a transmission of 2Gbps.

Besides the random topology, the authors examined the ring topology with connections established in a random manner each time a migration took place. Various migration strategies were implemented and their influence on the process of evolution was studied. The following one can serve as an example:

The user can decide if the subpopulation will be treated in one or more processes. One processor of the cluster system can treat one or more processes. A random communication topology is used in the migration process. An individual is moved with respect to user defined probability in a random position of randomly selected population. In the selected position being occupied by another individual, the later one migrates to the former position of the incoming individual. If the destination subpopulation is treated by the same process, it suffices a simple exchange. Otherwise, the individual is gathered in a message buffer together with the others willing to migrate from the current process. This message buffer is sent to all other processes which extract data corresponding to the incoming individuals and send back the data corresponding to the individuals being replaced. The algorithm stops when one of subpopulation satisfies the termination condition.

Authors conclude that the parallel execution on the cluster is able to speed-up the DE algorithm significantly and the global optimum is found with a higher probability even if there exist many similar sub-optimal solutions. Improvements in convergence were obtained even by sequential implementation due to the ability of the migration process to preserve the population diversity and

thus to avoid premature convergence. Efficiency of the described implementation of parallelised adaptive DE algorithm varies between 85% and 100%, depending on size of the population and used objective function.

5.2 Self-Organising Migrating Algorithm

Recently, group of evolutionary algorithms welcomed a new brave member – the Self-Organising Migration Algorithm - SOMA (Zelinka, 2002; Zelinka, 2004). It is very modern optimisation algorithm, which in most cases of test function benchmarks and also real-world optimisation tasks outperforms all other evolutionary-based optimisation techniques. Successful applications like active compensation of disturbing signals on a Langmuir probe measuring properties of RF-driven plasmas (Zelinka, 2004; Nolle et al., 2005), symbolic regression (Zelinka & Oplatková, 2005), a robot's trajectory optimisation (Oplatková & Zelinka, 2006), real-time deterministic chaos control (Zelinka, 2006), neural network synthesis (Zelinka & Volná, 2005), combustion engine optimisation, relay node placement in energy-constrained networks (Červenka & Zelinka, 2006) or aerodynamic optimisation of wing geometry (last three examples are thoroughly described in the Applications chapter) proved very high performance of this algorithm and predetermine SOMA to be employed in the most difficult assignments where the conventional optimisation methods cannot be used or other algorithms fail.

SOMA is based on the competitive-cooperative behaviour of intelligent creatures solving a common problem. Such behaviour of intelligent creatures can be observed anywhere in the world. A group of wolves or other predators may be a good example. If they are looking for food, they usually cooperate and compete so that if one member of the group is more successful than the previous best one (e.g. has found more food) then all members change their trajectories towards the new most successful member. It is repeated until all members meet at one food source. In SOMA, wolves are replaced by individuals. They 'live' in the optimized model's hyperspace, looking for the best solution. It can be said, that this kind of behaviour of intelligent individuals allows SOMA to realize very successful searches.

5.2.1 Serial SOMA

Because SOMA uses the philosophy of competition and cooperation, the variants of SOMA are called strategies. They differ in the way how the individuals affect the others. The basic strategy is called 'AllToOne' and consists of the following steps:

1 **Definition of parameters**. Before starting the algorithm, the SOMA parameters (popSize, Dim, PathLength, Step, PRT, Migrations, MinDiv) has to be defined. The user must also

create the specimen and the cost function that will be optimized. Cost function is a wrapper for the real model and must return a scalar value, which is used as a gauge of the position fitness.

2 **Creating a population**. New population with PopSize individuals is randomly generated.

3 **Migration loop**:

3.1 Each individual is evaluated by a cost function and the leader (individual with the best fitness) is chosen for the current migration loop.

3.2 All individuals except the leader perform their run towards the leader. The movement consists of jumps determined by the Step parameter until the individual reaches the final position given by the PathLength parameter. For each step of the individual, the PRTVector is recreated, the cost function for the actual position is evaluated and the best value is saved. Then, the individual returns to the position, where it found the best cost value on its trajectory.

3.3 New leader is chosen.

4 **Termination conditions test**. If the difference in cost values between leader and the worst individual is lower than value of the MinDiv parameter or the maximum of migration loops has been reached, the run of SOMA is terminated and the best position (the best set of parameters) is returned. In other case, the algorithm continues in step 3.

To get more information about this powerful evolutionary algorithm, please visit the SOMA's homepage at `http://www.fai.utb.cz/people/zelinka/soma/`.

5.2.2 Parallel SOMA

Inspired by models of evolutionary algorithms parallelisation summarized by Eric Cantú-Paz in his book (Cantú-Paz, 2000), considering implementation of already parallelised genetic and evolutionary algorithms, avoiding dead ends, including promising ideas and regarding our possibilities of employing parallel applications and taming CPU power in our university labs and classrooms, it appears that four feasible models parallelising the SOMA algorithm can be implemented.

Parallelisation at the objective function level (evolutionary algorithm running in one master process, evaluation of the objective function is migrated to subordinate processors) is not discussed here due to its high communication costs resulting in low efficiency.

Synchronous island model

This approach suits parallel SOMA running in the above described cluster platform very well. At each computation node, randomly initialized subpopulation are created according to configuration given by the master node (server). A node performs one SOMA migration (do not confuse with migration of individuals in parallel DE) and sends local leader to the server. After a migration loop is done on all terminals, server compares cost values of all received local leaders and chooses a global leader. This leader is then sent back to terminals and replaces the worst individual in local populations. This process is repeated until the termination conditions are satisfied.

As you can guess, the bottleneck of this implementation is the synchronism. A lot of CPU time of more powerful workstations is wasted by waiting for slow terminals to finish their migration loop. Due to heavy losses in computational performance, this approach was replaced by the next model. Nevertheless, only this parallelisation of SOMA could be fully compared to the simple serial version. As mentioned before, parallelisation can emerge higher robustness and performance to the world of evolutionary algorithms. If one want to compare evolutionary performance of serial and parallel version of the algorithm, this model of parallelisation must be used.

Asynchronous island model

To avoid time delays in the former parallelisation approach, synchronism of sharing and selecting the best individual was removed. After a terminal finishes its migration loop, local leader is sent to the server. Task of the master node is to maintain a global leader – every time it receives a leader from a subordinated node, it compares its cost value with the value of global leader and stores the better one. Consequently, the global leader is passed back to the terminal node where next migration loop is started. Again, this process is repeated until the stop conditions are fulfilled.

This parallelisation approach is also used outside the cluster platform. There was a need to use parallel SOMA on the Matlab and Mathematica platforms, but the cluster platform accepts only objective functions implemented in Java. Therefore a simple message passing interface enabling sharing a global leader among subpopulations was created (MPI, UDPOptCluster). The SOMA algorithm itself is implemented in the languages of Matlab and Mathematica and instantion of the UDPOptCluster Java extension is created before SOMA is started. In SOMA, after migration of all individuals is finished and a local leader determined, it is passed via the MPI to all other subpopulations. Then, a global leader is picked up from a local message box and next migration loop can follow. After the termination conditions are met, a message is sent to all other nodes to stop. Flow diagram of this process can be seen in Figure 5-1.

Usage of the UDPOptCluster extension for sharing a leader among subpopulations is limited only to local network as UDP packets (UDP multicasts are employed to deliver a leader to all participating computers) are not forwarded by routers present in the network. It means that this approach can be used only by limited number of computers.

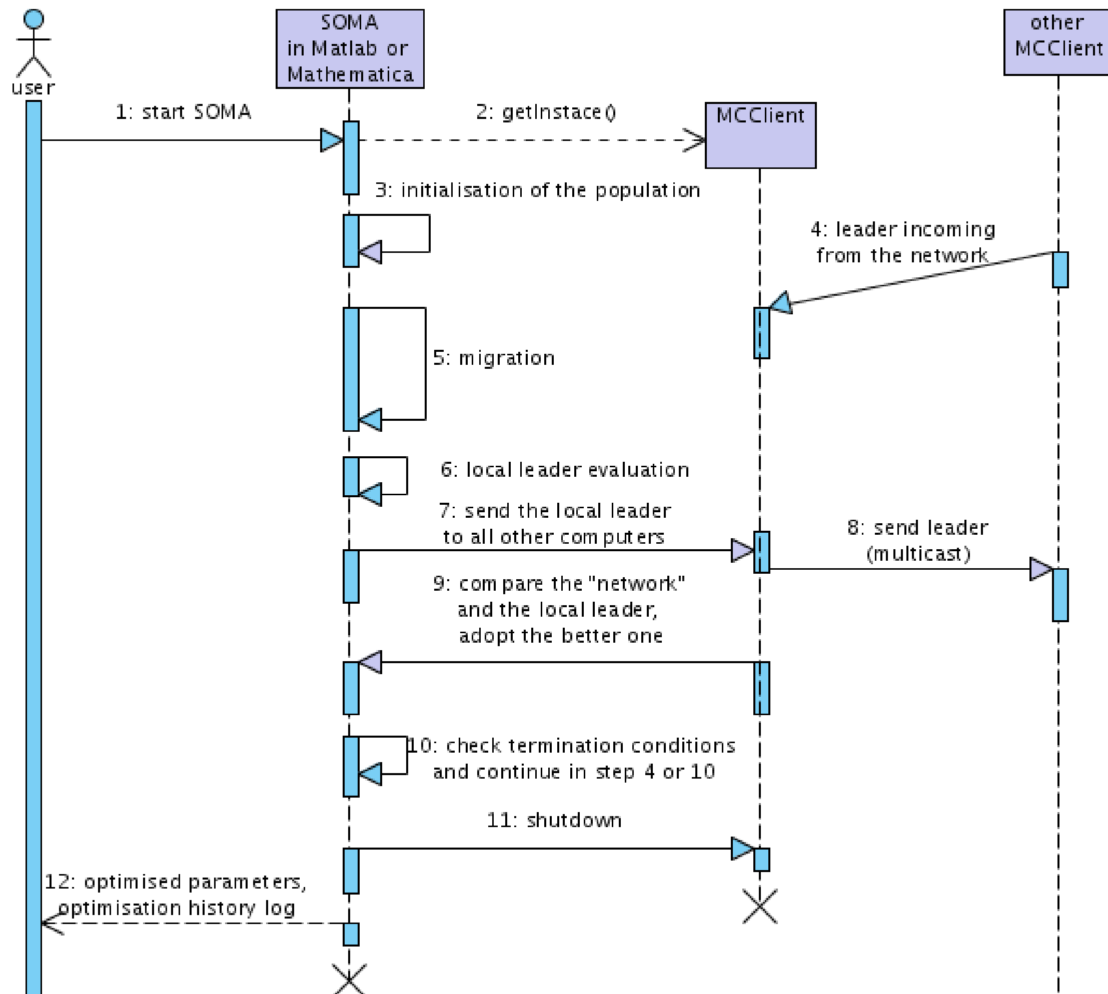


Figure 5-1: *Diagram of integration UDPOptCluster into SOMA implemented in Matlab or Mathematica and its communication with other terminals.*

Originally, the UDPOptCluster was implemented only as a helper when limited number of computers was needed to perform some optimisation tasks. Later, when experiments with openMosix (http://www.openmosix.org) and Mathematica were undertaken, UDPOptCluster served as a simple and useful MPI.

Asynchronous hybrid island - global population model

This approach combines a global population whose parts live on separate islands for some period of time. At the beginning of *somalisation* (optimisation using SOMA), a global population is created and randomly initialised. According to the number of available terminals, the population is divided into subpopulations by random selection of individuals (a portion of individuals is saved

aside to create a 'buffer' of not assigned individuals). These subpopulations are further distributed to subordinated processors. After a migration loop (or loops, there may be more than one) is performed, local population is sent back to the server where individuals are stored into their former positions in the global population. Then, another subpopulation of not used individuals (individuals that are not currently processed) is randomly assembled and the process is repeated. A surfeit of individuals ensures mixing of individuals when there is only one batch of not used individuals in the global population (otherwise identical subpopulation would be returned back to terminal).

Cellular model

The most advanced approach is the cellular model (also called as neighbourhood or diffusion model). Terminals of the cluster platform can create a virtual network – a grid, which is used for parallelisation of the SOMA algorithm. Information about position of a leader is passed asynchronously and not to all subpopulations at once.

Information about the position of the leader is spread over other terminals in steps, which is displayed in Figure 5-2. After a migration loop is finished (for example at the terminal in the middle marked with 0), local leader is passed to four neighbouring terminals (marked with 1) in the first step. After they finish their own migration, they compare the received leader to the local one. If the received leader is better than the local leader, it is adopted and replaces the worst individual in the population. Further, after next migration is done and in case the adopted leader still remains the best one in the population, it is passed again to neighbouring nodes in the second step (marked with 2). This communication strategy ensures that SOMA does not handle a single local solution, which results in more exhaustive exploration of the model hyperspace.

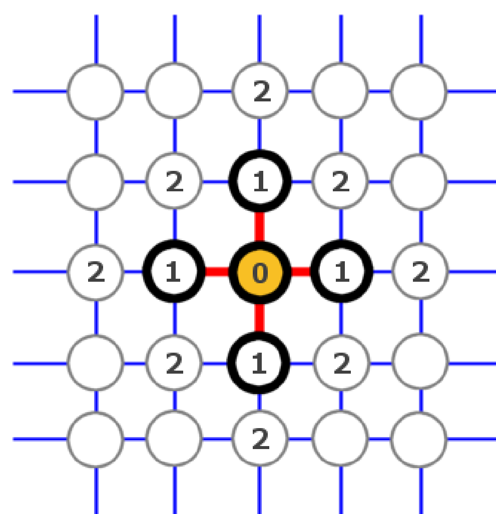


Figure 5-2*: Spreading of information in virtual network of terminals. Influence of a 'good' leader extends in steps across the entire grid.*

Information about local leader is not sent only to terminals in the neighbourhood but also directly to the server. Server continuously collects incoming candidate solutions and decides about termination of the search process. Normally, the stop condition in SOMA is based on comparison of the distance between the best and the worst individual. Here, as we cannot access all individuals to find the worse one, we do it in a different way: if there has not been observed any change in cost value of incoming leaders for some time, it could indicate that the optimisation process stopped in global (or local) extreme and is not going to continue. Then the termination condition is satisfied and we can stop the search process. The second option is to stop somalisation after given number of migration loops. As the master process knows how many terminals participate on the task, it can compute number of accomplished migration loops of the number of received leaders divided by the number of active processors.

5.2.3 Parallel SOMA performance

Parallelised SOMA was tested on teaching of a neural network. Described network configuration and training set were chosen only for this efficiency tests, they have no other meaning. SOMA configuration was: step = 0.11, pathLength = 3, prt = 0.3, minDiv = 0, migrations = 400. Number of modified neuron-wages = 315, population size = dim*2 = 630 individuals. The network had 4 layers with 3 neurons in the input layer, two hidden layers, 15 neurons each, and 3 neurons in the output layer. Linear transfer function was used in all neurons. The teaching set had 21 input and 21 output vectors and its structure can be seen in Table 5-1. Teaching process on single computer using SOMA lasted about 82 minutes (this test was executed on a workstation with configuration B (see the list of computers below)). Output listing of the teaching procedure can be seen in the same figure in the right column.

Regarding the time reduction of optimisation using SOMA, our parallel implementation proved very high performance and efficiency. Performance tests were executed on following computers:

Server:

- CPU AMD Athlon XP 1800+, 768 MiB RAM.

  Server did not perform any somalisation tasks, it only controlled the search process.

Workstations:

- 4 x Intel P4 2000 MHz, 256 MiB RAM (A)
- 8 x Intel P4 1800 MHz, 256 MiB RAM (B)

INPUTS			REQUESTED OUTPUTS			Neural network OUTPUTS		
Input 1	Input 2	Input 3	Output 1	Output 2	Output 3	Output 1	Output 2	Output 3
-1,00	-1,00	-1,00	-1,00	-1,00	-1,00	-1,00	-1,00	-1,00
-0,90	-0,90	-0,90	-0,90	-0,90	-0,90	-0,90	-0,90	-0,90
-0,80	-0,80	-0,80	-0,80	-0,80	-0,80	-0,80	-0,80	-0,80
-0,70	-0,70	-0,70	-0,70	-0,70	-0,70	-0,70	-0,70	-0,70
-0,60	-0,60	-0,60	-0,60	-0,60	-0,60	-0,60	-0,60	-0,60
-0,50	-0,50	-0,50	-0,50	-0,50	-0,50	-0,50	-0,50	-0,50
-0,40	-0,40	-0,40	-0,40	-0,40	-0,40	-0,40	-0,40	-0,40
-0,30	-0,30	-0,30	-0,30	-0,30	-0,30	-0,30	-0,30	-0,30
-0,20	-0,20	-0,20	-0,20	-0,20	-0,20	-0,20	-0,20	-0,20
-0,10	-0,10	-0,10	-0,10	-0,10	-0,10	-0,10	-0,10	-0,10
0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
0,10	0,10	0,10	0,10	0,10	0,10	0,10	0,10	0,10
0,20	0,20	0,20	0,20	0,20	0,20	0,20	0,20	0,20
0,30	0,30	0,30	0,30	0,30	0,30	0,30	0,30	0,30
0,40	0,40	0,40	0,40	0,40	0,40	0,40	0,40	0,40
0,50	0,50	0,50	0,50	0,50	0,50	0,50	0,50	0,50
0,60	0,60	0,60	0,60	0,60	0,60	0,60	0,60	0,60
0,70	0,70	0,70	0,70	0,70	0,70	0,70	0,70	0,70
0,80	0,80	0,80	0,80	0,80	0,80	0,80	0,80	0,80
0,90	0,90	0,90	0,90	0,90	0,90	0,90	0,90	0,90
1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00

Table 5-1: *Training set for neural network and its real output used as a time-demanding objective function example for the process of somalisation*

Computers were interconnected by the 100BASE-TX LAN. As there were two different types of processors, they were added into the cluster in order to obtain linear increase in computation power. It means that for 4 CPUs we employed processors ABBB, for 8 CPUs processors AAABBBBB and finally, for 10 CPUs we used processors in the AAABBBBBBB configuration. Tests for every cluster configuration were executed 10 times, average efficiency of tested parallelisation strategies running on various cluster configurations can be seen in Table 5-2.

Synchronous and asynchronous island models and cellular model were tested using the author's Java cluster platform with neural network as an objective function. As the time constants were set very low, maximal time losses caused by the platform were limited to 4 seconds.

For UDPOptCluster MPI experiment, manually executed Matlab with SOMA calling the "Glauert2_cf" cost function (see the "Aerodynamic optimisation of wing geometry" chapter in Applications below) were used. SOMA configuration was: step = 0.11, pathLength = 2.4, prt = 0.1, minDiv = 0, migrations = 45, population size = dim*4 = 60 individuals. 35.8 minutes are normally required to perform this optimisation task on a computer in configuration A.

Synchronous island model					
	Serial SOMA	Parallel SOMA			
Number of processors	**1**	**4**	**8**	**10**	**12**
Average processing time [s]	4920	1278	659	535	449
Ideal processing time [s]	4920	1230	615	492	410
Time loss [s]	0	48	44	43	39
Efficiency [%]	100,00	96,24	93,32	91,96	91,31

Asynchronous island model					
	Serial SOMA	Parallel SOMA			
Number of processors	**1**	**4**	**8**	**10**	**12**
Average processing time [s]	4920	1251	634	512	429
Ideal processing time [s]	4920	1230	615	492	410
Time loss [s]	0	21	19	20	19
Efficiency [%]	100,00	98,32	97,00	96,09	95,57

Cellular model					
	Serial SOMA	Parallel SOMA			
Number of processors	**1**	**4**	**8**	**10**	**12**
Average processing time [s]	4920	1236	621	499	418
Ideal processing time [s]	4920	1230	615	492	410
Time loss [s]	0	6	6	7	8
Efficiency [%]	100,00	99,51	99,03	98,60	98,09

UDPOptCluster					
	Serial SOMA	Parallel SOMA			
Number of processors	**1**	**4**	**8**	**10**	**12**
Average processing time [s]	2310	595	307	250	209
Ideal processing time [s]	2310	578	289	231	193
Time loss [s]	0	18	18	19	17
Efficiency [%]	100,00	97,06	94,06	92,40	92,11

Table 5-2: *Overview of parallel SOMA test results*

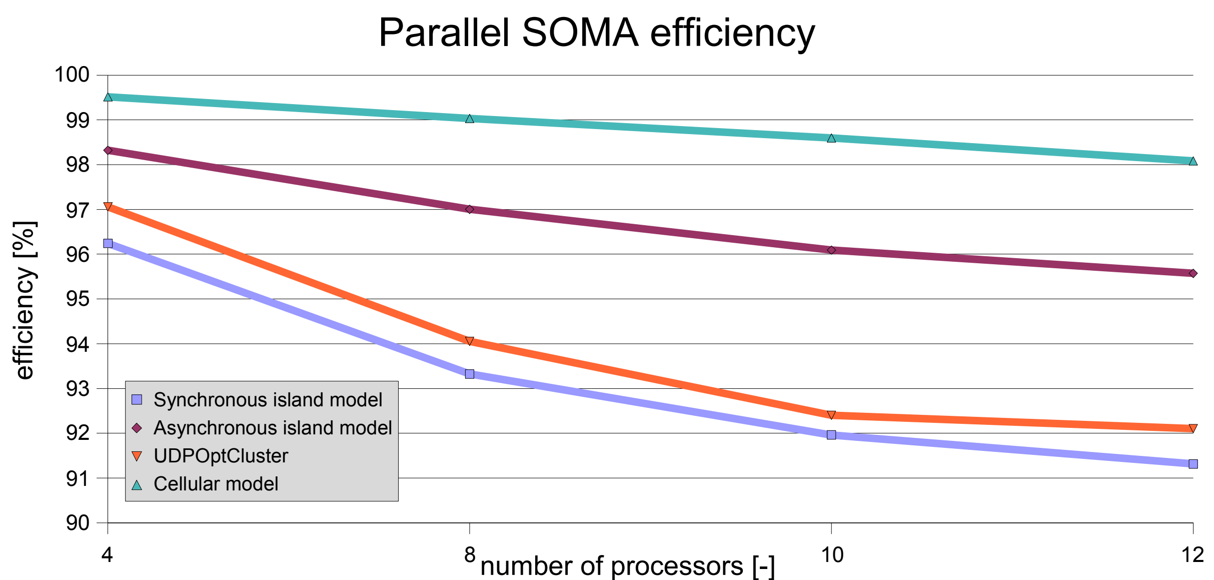


Figure 5-3*: Performance comparison of various parallelisation models*

As we can see from Table 5-2, the drop in efficiency is mostly caused by almost constant time-loss. This (especially visible at the cellular model) might be influenced by the overall cluster platform efficiency. As it was expected, the synchronous island model showed relatively low productivity, which has been significantly increased in its asynchronous alternative. Very promising is the cellular model; its efficiency is on the edge of the cluster framework efficiency. On the other hand, we must admit that the UDPOptCluster approach was a little less efficient than we expected, but considering the fact that it is just an ad-hoc instant cluster substituting a real parallel machine, it showed rather decent performance. However, it is good to remind that these simulation test were performed under almost ideal conditions. In real deployment situation, the efficiency will be definitely lower, decreased by disturbances like students rebooting or turning off computers used as terminals or other accidents happening in computer networks every day.

5.3 Chapter Summary

In this chapter two modern and very powerful evolutionary algorithms, the Differential Evolution (DE) and the Self-Organising Migrating Algorithm (SOMA), are presented. In the first section, the principle of Differential Evolution (DE) and its parallelisation strategies are described. The second section concerns with the parallel SOMA implementation. Four parallelisation strategies inspired by approaches used in parallel DE and other parallel genetic/evolutionary algorithms are exhaustively analysed and results of benchmark tests are provided (see Table 5-2 and Figure 5-3). Based on the obtained results, we can claim, that the Cellular model is the most efficient parallelisation strategy of the SOMA algorithm.

Search for any algorithmic improvements of SOMA brought by parallelisation was not successful. All investigated modifications of SOMA with the aim to increase its robustness or performance failed. This opens a question: Might the current design of SOMA be the most optimal one?

6 APPLICATIONS

This chapter thoroughly describes three realized applications of SOMA algorithm. Organisation of all section is as follows: First, the used theory is introduced, then an existing reference example is analysed (if available) and finally, the particular solution to given problem is presented.

6.1 Combustion engine optimization

Optimisation of a modern four-cylinder engine was an application where parallel SOMA demonstrated its high performance for the first time. This project was accomplished in cooperation with Department of Electric and Electronic Engineering (DEEE) at Strathclyde University in Glasgow, UK.

Model of the engine was created by Simon Flint for his dissertation (Flint, 2004) as a model of a real engine in cooperation with the Visteon company (see Figure 6-1). The author's primary aim of making this model was the application of twin independent variable camshaft timing (TIVCT) to the engine. Because this model covers the entire engine, it was also used at DEEE for controller design for valves, cams and other engine equipment.

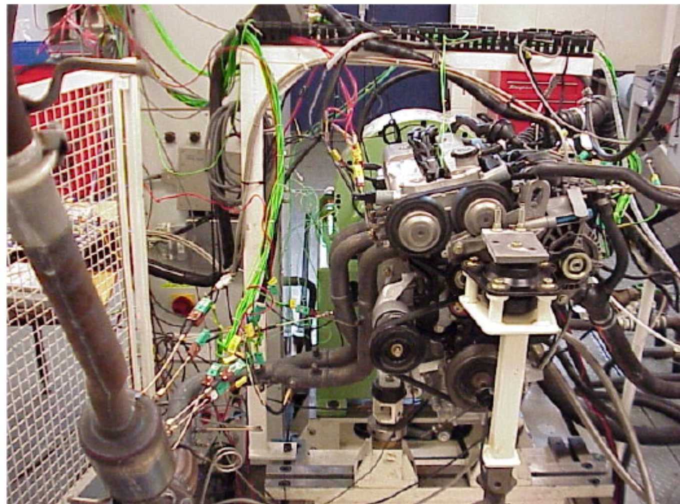


Figure 6-1: *TIVCT engine installation in Visteon*

The optimisation task was to minimise three output values by modification of input values for certain number of set point combinations of other two input values. Specifically, the input values were:

- **MAP** – manifold pressure (optimised),
- **IV** – inlet valve timing (optimised),
- **EV** – exhaust valve timing (optimised),

- **SA** – spark advance (optimised),
- RPM – revolutions per minute (set point),
- TQ - torque (set point).

Set of output values which had to be minimised was:

- BSFC – break specific fuel consumption,
- BSNOx – break specific NOx,
- TQError = (current TQ – desired TQ) – deviation from the desired torque.

The optimised model was created in Matlab Simulink, consisting of neural-networks, fuzzy blocks and lookup tables. A first level of block schema can be seen in Figure 6-3. Computation of the cost function was done using

$$costValue = abs(BSNOx)\cdot 35 + abs(BSFC) + abs(TQError)\cdot 1000 + (60 - INdeg + EXdeg)\cdot 2 + pen \ ,$$

where INdeg and EXdeg are positions of inlet and exhaust valves and *pen* stands for penalisation, which was applied when the model output values of TQ and/or BSFC were (surprisingly) negative.

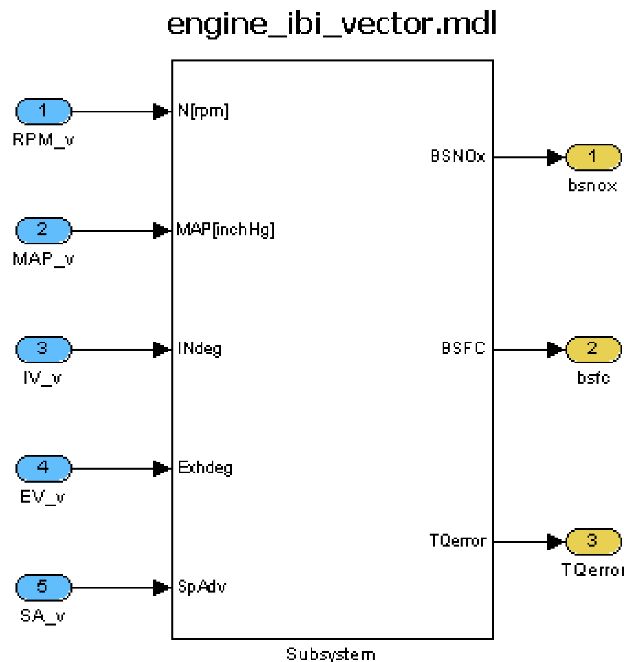


Figure 6-2: *Block encapsulating model of the optimised engine and representing an interface between SOMA and the model*

As mentioned above, optimal input values were searched for combination of RPM and TQ set points. RPM values were in a range from 500 to 6000/min, torque between 10 and 120 Nm. An example of set points, found input values and corresponding outputs can be seen in Table 6-1.

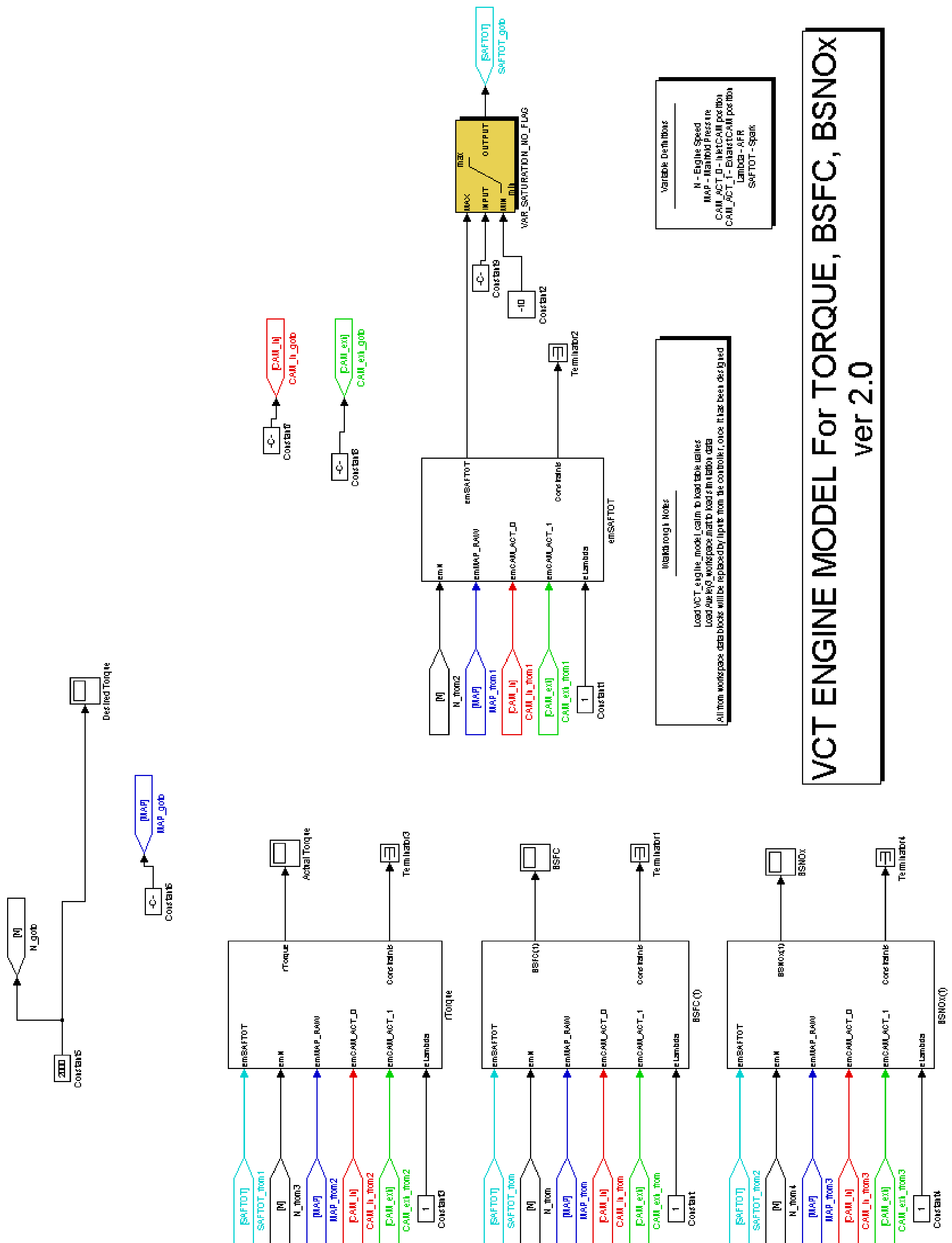


Figure 6-3: *Block schema of the optimised engine*

RPM	Torque	MAP	IV	EV	SpAdv	CostValue	BSNOx	BSFC	Tqerror
500	10	12,1888	20,1688	46,2981	45,5170	246,8691	-2,1102	0,5793	0,0004
500	20	12,0884	24,1299	42,1244	46,6889	156,0896	-0,0024	0,0005	0,0000
500	30	12,0949	36,5190	42,1288	59,9361	172,9437	1,0944	2,7485	0,0026
500	40	26,2241	49,4581	48,3869	-3,7326	240,1321	3,4634	0,7497	0,0015
500	50	26,0733	49,8631	44,5849	-0,9131	251,1632	3,9715	1,2059	0,0020
500	60	26,1523	49,8146	39,2663	1,1032	268,1833	4,7665	2,2918	-0,0003
500	70	26,3406	49,5339	32,5074	2,7908	293,9875	5,9224	0,7270	0,0000
500	80	26,7177	49,9297	34,5595	9,5090	331,4693	6,7618	5,4677	-0,0009
500	90	27,5408	49,4967	20,1155	12,0051	372,2903	9,4926	2,1376	2,2946
500	100	29,3945	49,5294	23,2436	16,8318	424,0615	11,0167	12,5407	2,4463
500	110	29,5251	41,1297	23,2335	14,4847	505,1512	12,0200	0,1709	0,0002
500	120	26,4801	28,4863	20,7871	21,1926	3684,9902	15,6619	216,7096	-20,7176
1000	10	12,2488	20,1408	47,2222	44,2162	180,0141	-0,1484	1,2612	-0,0003
1000	20	12,2382	30,9918	47,5156	50,2636	155,2721	0,0589	0,1291	0,0000
1000	30	28,1479	46,9693	46,8612	-9,3026	232,7321	3,0121	0,3172	-0,0001
1000	40	27,3354	49,9627	46,6416	-4,5161	231,8721	3,3268	1,7913	-0,0300
1000	50	27,5820	47,3744	43,4613	-3,2523	252,2918	3,9781	0,4432	-0,0007
1000	60	27,5108	49,9451	37,5271	-0,7858	270,4861	4,8937	0,2816	-0,0038
1000	70	27,3602	49,8423	38,5068	4,4365	297,1102	5,6631	0,5309	0,0052
1000	80	27,4812	49,3997	20,0497	7,6334	328,0391	8,9059	17,8115	7,4478
1000	90	28,2360	49,4195	28,5105	9,4624	372,9062	8,3505	6,2340	0,0541
1000	100	29,3730	49,7832	24,8585	12,7069	417,9270	10,4588	13,8034	2,4483
1000	110	29,5266	40,8831	21,5395	11,4348	509,2096	12,1202	3,8509	0,0060
1000	120	29,5299	25,8802	20,0163	17,2928	787,6198	16,8411	89,8944	0,0000
1500	10	12,0240	21,4523	49,8313	45,0762	229,4324	1,4376	3,2905	0,0019
1500	20	12,0373	35,7789	47,8893	53,6662	238,3804	2,6816	0,2006	0,0010
1500	30	29,0190	48,1777	49,1947	-8,4678	216,1141	2,6518	1,2077	-0,0004
1500	40	28,8243	49,9044	45,6979	-6,0843	223,2676	3,1163	3,3876	-0,1916
1500	50	29,0846	49,5542	41,5629	-4,8110	238,8258	3,8315	0,3059	0,0032
1500	60	29,2317	49,2013	36,3140	-3,6610	262,3410	4,8260	4,1808	0,0023
1500	70	28,8331	49,7221	38,2962	1,5571	290,5440	5,4898	1,0760	0,0002
1500	80	29,1182	49,4216	33,4735	3,2242	325,3320	6,6532	2,0766	-0,0359
1500	90	29,4487	48,9724	28,0296	5,0887	359,6934	8,0343	1,1349	0,0067
1500	100	29,4805	48,2777	23,8334	9,0630	432,6704	9,7019	21,4747	-0,0068
1500	110	29,5116	34,0558	20,2633	9,9772	529,8172	14,6644	24,8498	5,5274
1500	120	29,5299	21,1603	20,1707	11,3552	682,8903	15,3465	27,5995	-0,0001
2000	10	25,7217	49,4614	48,7244	-9,9442	366,8758	0,6724	222,1708	0,0075
2000	20	28,9986	49,8778	49,9802	-9,9072	255,8419	1,8291	69,3322	0,0024
2000	30	29,1847	49,9874	49,5550	-7,6043	244,3468	2,2913	45,0363	0,0003
2000	40	29,4839	49,6395	46,0361	-6,4386	253,0077	2,7976	41,2504	0,0009
2000	50	29,5294	48,0556	49,0982	-3,1174	255,6044	3,4344	11,5742	0,0000
2000	60	29,4378	47,9684	47,9660	-0,2261	278,6599	4,2312	10,3137	0,0003
2000	70	29,4562	48,9305	46,3120	3,1576	308,9284	5,2858	8,3849	0,0107

Table 6-1: *Output of the engine optimisation process*

6.2 Relay node placement in energy-constrained networks

Due to the fast progress in electronics and electromechanics, wireless networks based on a large number of inexpensive miniature devices with sensing, computing and communication capability are comming to everyday life.

One possible application of these networks is a weather monitoring sensor array spread over a large geographic area. Data generated from all points of the sensor field is gathered using multi-hop communication to a base station for further processing.

The weak point of this network is the limited power source capacity of single nodes that are powered by batteries. A number of papers has already been concerned with optimal sensor placement (Dasgupta, Kukreja & Kalpakis, 2003), energy efficient routing designs (Chu, Haussecker & Zhao, 2002; Krishnamachari & Ordóñez, 2003) and protocols with the objective of maximizing lifetime (Bhardwaj & Chandrakasan, 2002) or amount of transmitted data.

The aim of this work was to develop a method for finding optimal positions for a certain number of relay nodes to maximise the network lifetime. To reach this, the Self-Organizing Migration Algorithm (SOMA) was used on message routing models described in (Falck et al., 2004; Floréen et al., 2005).

This project was based on cooperation with the Laboratory for Theoretical Computer Science at Helsinki University of Technology, Helsinki, Finland.

6.2.1 Network model and balanced data gathering as a flow LP

The authors of (Falck et al., 2004) consider a network consisting of n sensor nodes, m relay nodes and a base station, all with predetermined locations except the relay nodes whose locations may be changed. The set of all nodes is denoted as $V = B \cup S \cup R$, where B, S and R represent the sets consisting of the base station node, sensors and relays.

Each node $i \in V$ has an initial energy supply of e_i units. The sink node is handled as a special case with $e_i = \infty$. The mission of the network is to gather data generated at the sensor (source) nodes to the base station (sink) node under their energy constraints during the desired operation time T.

It is assumed that the sensors generate data asynchronously and in such small unit packets that the process can be modelled by assigning an *offered data rate* parameter s_i, $i \in S$, to each sensor node. The energy cost of forwarding a unit of data from node i to node j is given by a parameter d_{ij} and the cost of receiving a unit of data is given by a parameter c. It is also assumed that the transmission rates are low enough, so that collisions and signal interferences can be ignored in the model.

The model places no restriction on the values of the parameters d_{ij} and c. In the commonly used simple radio-link, models d_{ij} would be taken to be proportional to $c_t + D_{ij}^{\alpha}$, where c_t corresponds to the energy consumed by the transmitter electronics and D_{ij}^{α} corresponds to the energy consumed by the transmit amplifier to achieve an acceptable signal-to-noise ratio at the receiving node. D_{ij} is the physical distance between nodes i and j and the exponent $\alpha \in$ <2, 4> models the decay of the radio signal in the ambient medium. The cost c corresponds to the energy consumed by the receiver electronics.

A flow variable f_{ij} indicates the rate of data forwarded from node i to node j, the energy constraints in the network can be expressed as $\left(\sum_j d_{ij} f_{ij} + \sum_j c f_{ji}\right) \cdot T \leqslant e_i$ for all $i \in V$. Because of the energy constraints in the network, the sensors cannot usually achieve their full offered data rates productively. Therefore variable r_i indicates the actual achieved data rate at each sensor. One goal of the data flow design for the network is to maximise the total, or equivalently, the average achieved data rate $(1/n)\sum_{i \in S} r_i$. However, taking this as the singular objective might lead to the "starvation" of some sensor nodes. Typically, the average data rate objective is maximised by data flows that only forward data generated close to the sink and do not allocate any energy towards relaying data generated at distant parts of the network.

To counterbalance this tendency, there is a *minimum achieved rate* variable l, with the constraint $r_i > l$ for all $i \in S$, which is to be maximised simultaneously with the average data rate. The trade-off between these two conflicting objectives is determined by a parameter λ, , where value $\lambda=0$ gives all weight to the average achieved rate objective and value $\lambda=1$ to the minimum achieved rate objective. The combined objective F_{λ} is called the *balanced data rate* or *balanced rate*.

Different sensors may submit different types of data. At each unit of time, the average amount of data transmitted from one sensor might be one bit and from another sensor ten bits; however, the one bit may be equally valuable for the application as the other sensor's ten bits. As a generalisation, weights *wi* are assigned to the data rates from different sensors according to their importance. A natural choice is *wi = 1/si*, which normalises the data rates of all sensors to the interval [0, 1] and expresses the idea that an equal proportion of each sensor's offered data should be transmitted. For simplicity, this model uses equal offered data rates and equal weights for all types of communication.

The variable f_{ij} models the flow of data from i to j. Variable q_i models the quantity of data generated at source i and ultimately received at the sink. Described model can be formulated as the following

linear program, which can then be solved using standard techniques.

$$\text{Maximise} \quad F_\lambda := (1-\lambda)\frac{1}{n}\sum_{i\in S} w_i r_i + \lambda l$$

$$F_\lambda := (1-\lambda)\operatorname{avg}_{i\in S} q_i + \lambda \min_{i\in S} q_i \qquad (1)$$

$$\text{subject to} \quad \sum_{j\in V} f_{1j} = 0$$

$$\sum_{j\in V} f_{ij} = r_i + \sum_{j\in V} f_{ji} \qquad i\in S$$

$$\sum_{j\in V} f_{ij} = \sum_{j\in V} f_{ji} \qquad i\in R$$

$$\sum_{j\in V} Td_{ij} f_{ij} + \sum_{j\in V} Tcf_{ji} \leqslant e_i \qquad i\in V$$

$$r_i \leqslant s_i \qquad i\in S$$

$$w_i r_i \geqslant l \qquad i\in S$$

$$f_{ij} \geqslant 0 \qquad i, j\in V$$

$$f_{ii} = 0 \qquad i\in S$$

A flow matrix f_{ij}, obtained as a solution of this linear program, can be used to route approximately r_i unit-size data packets from each source node i in S to the sink node 1, assuming that all the r_i and f_{ij} values are large. At each node k, simply forward the first $[f_{k1}]$ packets to node 1 (the sink), the next $[f_{k2}]$ packets to node 2, the next $[f_{k3}]$ packets to node 3 and so on. A little more elegant solution is to randomise the routing strategy so that each incoming packet at node i is forwarded to node j with probability $f_{ij} / \sum_k f_{ik}$.

6.2.2 The effect of relay nodes

The performance of the sensor network can be improved by augmenting the network by a number of auxiliary relay nodes. Unlike sensor, whose locations are assumed to be predetermined, the locations of the relay nodes may be chosen to optimise the network performance. The relay nodes do not generate data themselves, they are entirely oriented on forwarding data to other nodes in the network. Relay nodes are supposed to be powered by stronger energy source than sensor nodes.

Grid and incremental relay node placement methods

In the case of a square area covered by sensor nodes, a straightforward method to place $m = k^2$ relay nodes is to position them in a regular k x k grid inside the square.

The authors of (Falck et al., 2004) suggest other solution to the relay node placement problem. Their algorithm performs a multidimensional search in the following manner. Given a starting point y, a suitable direction d is first determined and then the flow problem is optimised in this direction

by performing a line search. Thereafter, a new direction *d'* is chosen and, again, the flow problem is optimised starting from the previous optimum in the direction *d'*. The process is repeated until a good solution is found, or the algorithm converges to a (possibly) local optimum. The starting point *y1* is chosen as the centre of mass of the offered data rates s_i and achieved data rates r_i.

The idea is to place a new relay node initially in a region of the network where the achieved data rates r_i are small compared to the offered rates s_i. It is reasonable to think that the ideal location of the node would be, at least with high probability, somewhere between this region and the sink. Therefore the first search line is chosen in direction of the sink node. This idea is extended to determine the remaining search directions as the algorithm proceeds. The search directions are chosen pairwise: in the direction of the sink node and orthogonal to it. Line searches can, in principle, be performed by almost any standard one-dimensional search method, the main limiting factors are the complexity and possible roughness of the objective function F_λ.

Relay node placement using SOMA

Relay node placement represents an optimisation problem which suits SOMA very well. The model described above handles with an objective function (1), which represents a counterbalance between two objectives – to maximise the average quantity of data and also to maximise the minimum quantity of data gathered from the network.

In this optimization task, the modified parameters of the model are the positions of relay nodes within the area covered by sensor nodes, which means that we search for relay nodes coordinates. Therefore, the structure of SOMA's individual is defined as:

$$individual_i = \{x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_n,\},\ \ i \in \langle 1, popSize \rangle,$$

where *n* is number of relay nodes and *popSize* the number of individuals in population.

The cost function for SOMA returns negative value of the F_λ objective function (negative because SOMA is searching for minimal value of the cost function but the objective function should be maximised).

For both simulation experiments, the network settings were as follows. Sensor nodes were placed in a square area with the length of the edge 1 km. The sink was located at the centre of the southern side. While sensor nodes had energy of 20 J, the relay energy constraint was set to 2000 J. These settings are equal to those used in (Falck et al., 2004). All simulations were performed with the balancing factor λ set to 0.5.

The aim of the first test was to compare and contrast the results of relay node placement to a regular

grid of sensors (see Figure 6-4) provided by SOMA versus results generated by the incremental relay node placement algorithm presented in (Falck et al., 2004). In fact, this step was aimed more on verification of the correctness of node placement by SOMA and to confirm the conjecture that value of the objective function of regular grid network might depend on the amount of energy brought into network with the relay nodes linearly.

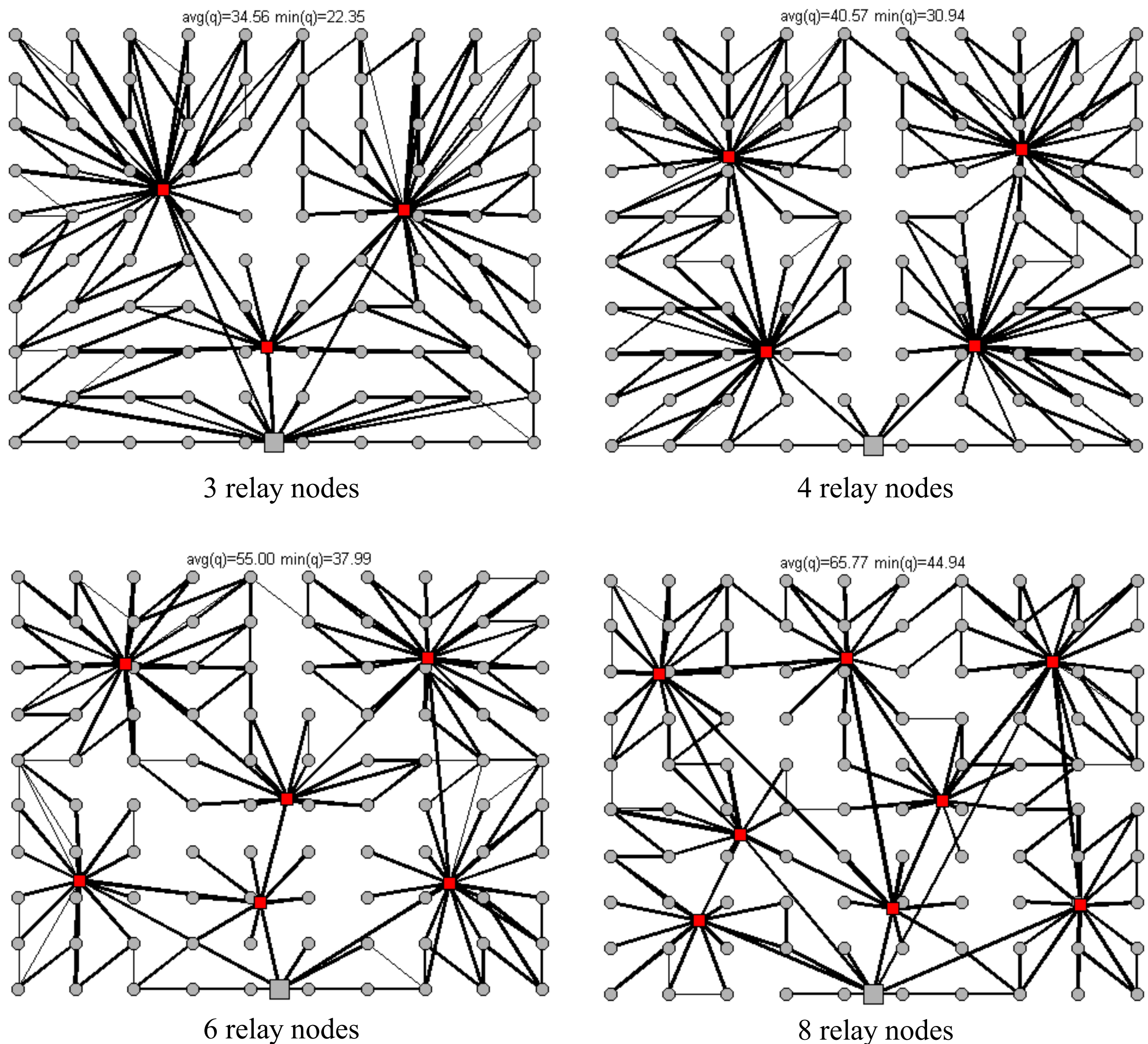


Figure 6-4: *Examples of relay node placement experiments to a regular grid of sensors. Gray circles represent sensor nodes, small red squares are relay nodes and the large grey node at the south side of the area is a sink where all data is collected*

As you can see in Figure 6-5, SOMA provided slightly better results than the incremental relay node placement algorithm presented in (Falck et al., 2004). Variances among positions of relay nodes seem to be based more on influence of rounding errors than on differences in algorithmic performance. Since solving this task using SOMA was rather demanding and at that time there were

no spare computers exploitable for parallel computations, experiments were performed only up to 6 relay nodes.

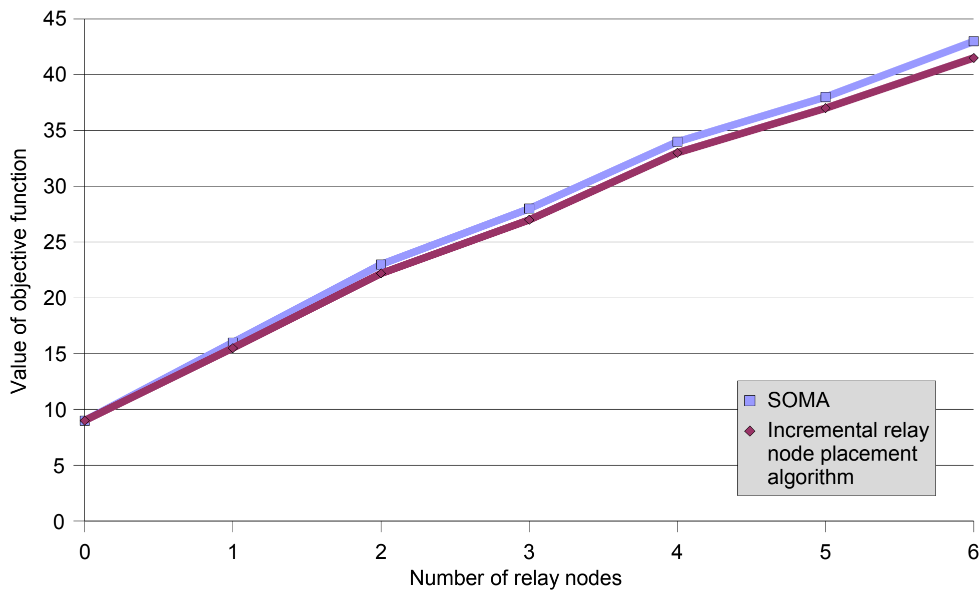
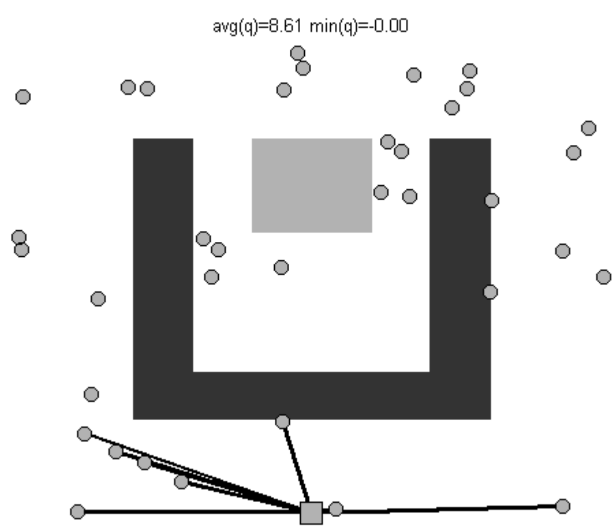


Figure 6-5: *Balanced data rate $F_{0.5}$ as a function of the number of relay nodes. Results for array with 100 source nodes placed in regular grid 10x10 without any obstacles in the landscape*

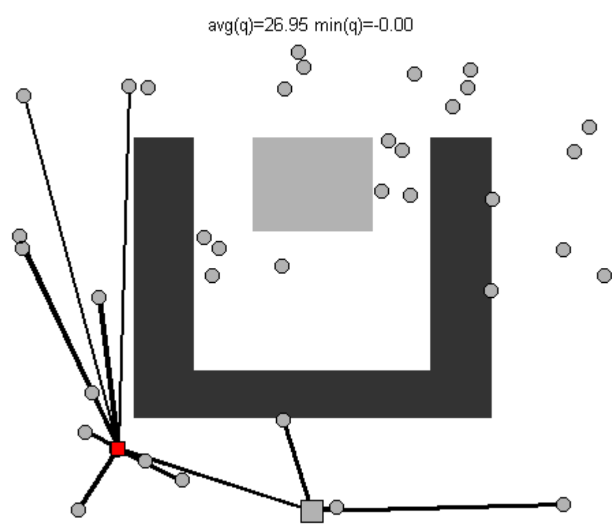
In the second experiment, the relay node placement by SOMA on network of 36 randomly placed sensor nodes was tested. To make this task a little more complicated, two obstacles were placed into the landscape. While the first type of obstacle represents a building - an impenetrable obstacle for signal, the second one is a lake, which the radio transmission can overcome. No node can be placed into both types of obstacles.

Simulation results for varied number of relay nodes (from 0 up to 10 relays) can be seen in Figure 6-6. On the top of every network schema, you can see components $avg(q_i)$ and $min(q_i)$ of the balanced data rate F_λ objective function. The aim was to maximise both these components.

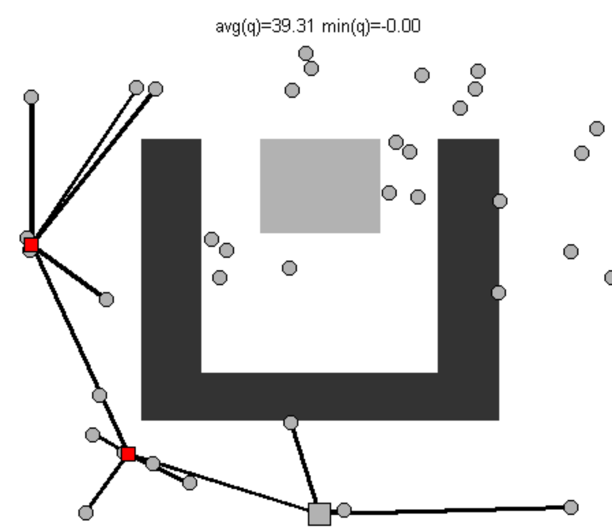
Figure 6-7 shows values of the objective function for experiments with 1 to 10 nodes. Found positions do not have to be the true optimal ones, but we can say that this is the best solution for used SOMA settings and number of iterations. Although SOMA appears to be very robust algorithm, it can happen that the process of searching for global extreme can stop in a local solution.



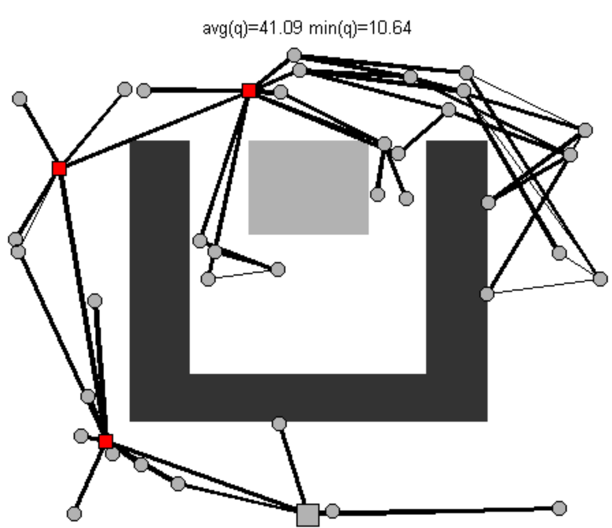
0 relay nodes



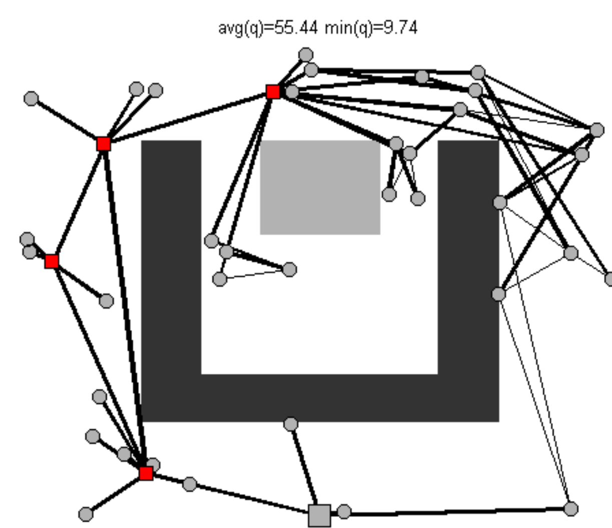
1 relay node



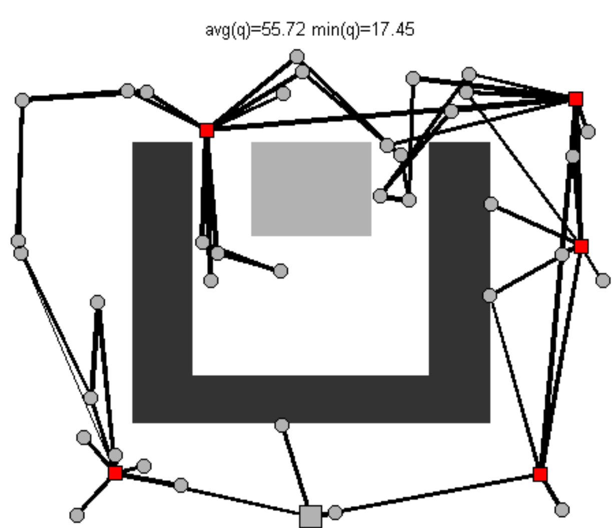
2 relay nodes



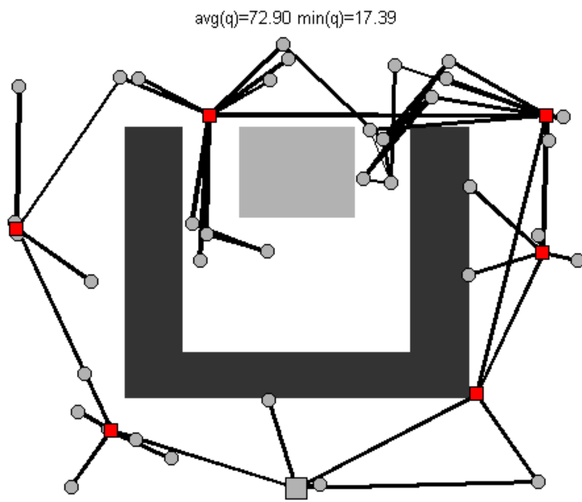
3 relay nodes



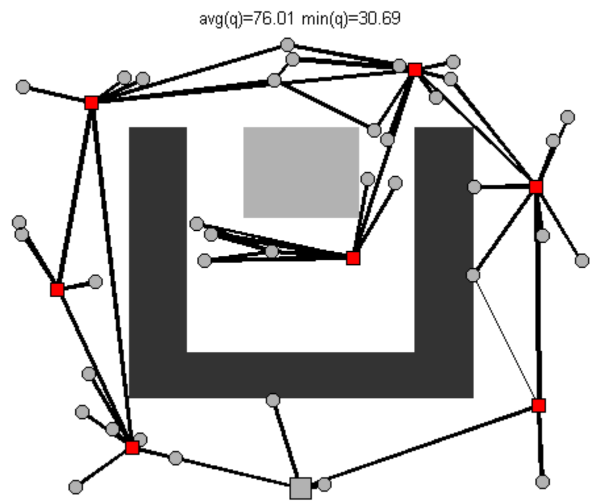
4 relay nodes



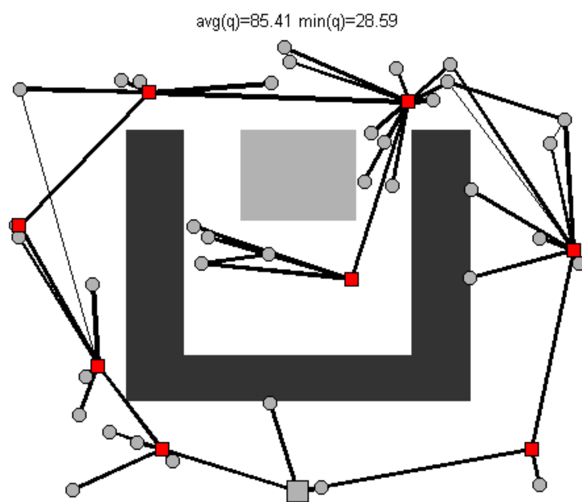
5 relay nodes



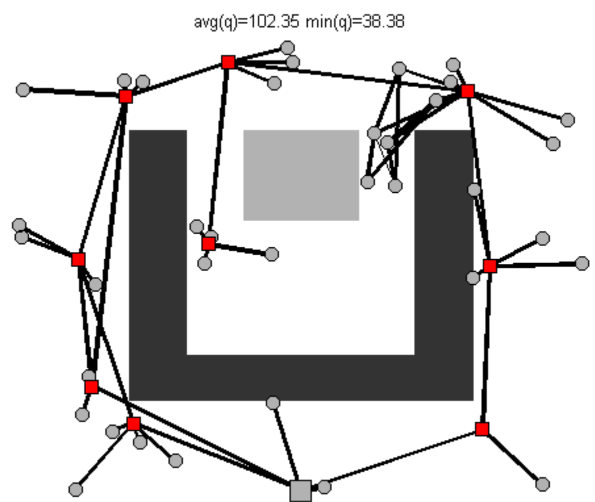
6 relay nodes



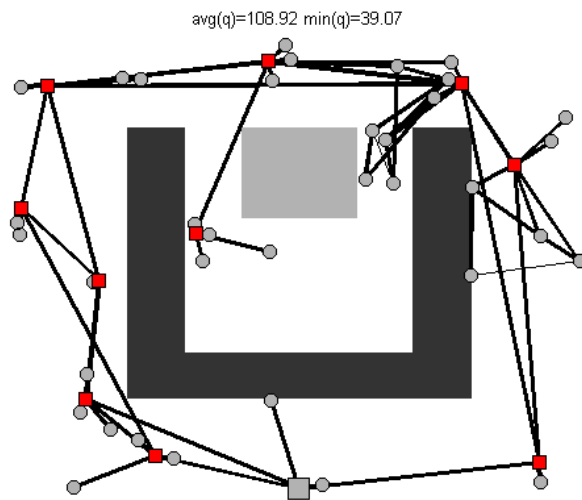
7 relay nodes



8 relay nodes



9 relay nodes



10 relay nodes

Figure 6-6: *Examples of relay node placement experiments on a network with 36 randomly placed sensor nodes. Gray circles represent sensor nodes, small red squares are relay nodes and the large grey node at the south side of the area is a sink where all data is collected.*

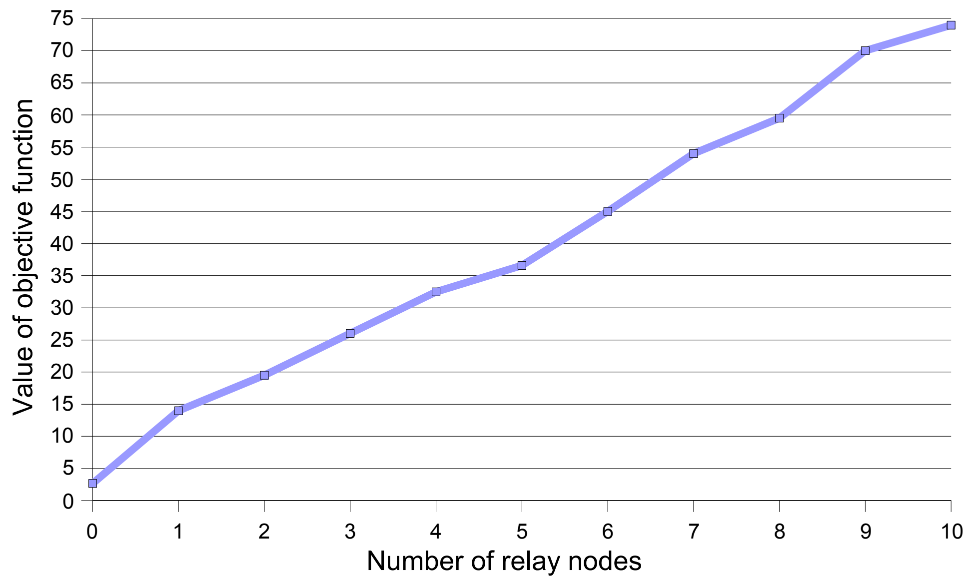


Figure 6-7: *Balanced data rate $F_{0.5}$ as a function of the number of relay nodes. Results for array with 36 randomly placed source nodes with obstacles in the landscape.*

To conclude this, new pioneering method offering an alternative solution to the problem of relay node placement using evolutionary algorithms was developed. In cases, where ordinary techniques can be insufficient, evolutionary approach might help. Results obtained from performed simulations proved that employing evolutionary algorithms for this type of problem can be a good choice.

6.3 Aerodynamic optimisation of wing geometry

Development of an aerodynamic shape optimisation methods is important to improve design efficiency in today's competitive environment for the commercial aircraft industry. Aerodynamic-wise optimal shape of an aircraft does not only delight an expert's eye, it is a crucial factor affecting the plane performance and thus its success on the world markets.

In this chapter, we focus on optimisation of a wing geometry. To optimise a shape of a wing does not only mean to reduce its drag and maximise lift. There are also other factors that influence the fitness of a wing for a particular aeroplane. Results presented here are based on collaboration with the Evektor company, the Department of Aerodynamics. Theoretical part of this section is based on (Florián, 1963; Houghton & Carpenter, 2003) and stimulating discussions with enthusiastic aerodynamicists.

6.3.1 The vortex system theory

Based on the Zhukovsky theorem, influence of an infinite wing put into a flow of liquid on this liquid can be substituted by influence of a potential vortex. The vortex system can be divided into three main parts: the starting vortex, the trailing vortex and the bound vortex system (the last one is

also denominated as a lifting vortex). Each of these may be treated separately, but they all are component parts of one whole system. The total vortex system associated with a wing form a complete vortex ring that satisfies all physical laws. The starting vortex, however, is soon left behind and the trailing pair stretches effectively to infinity as steady flight proceeds. For practical purposes the system consists of the bound vortices and the trailing vortex on either side close to the wing. This three-sided vortex has been called the *horseshoe vortex* (Figure 6-8).

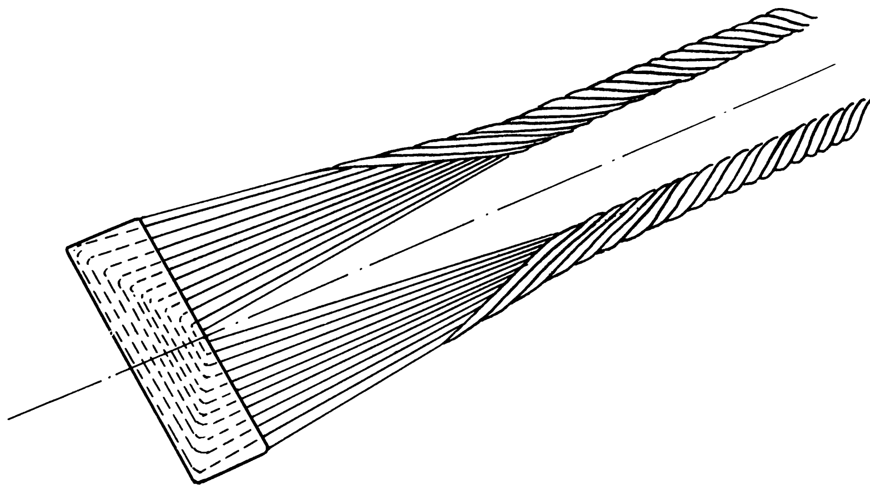


Figure 6-8: *The horseshoe vortex*

Study of the completely equivalent vortex system is largely confined to investigating wing effects in close proximity to the wing. For estimation of distant phenomena the system can be simplified to a single bound vortex and trailing pair, known as the simplified horseshoe vortex (Figure 6-9).

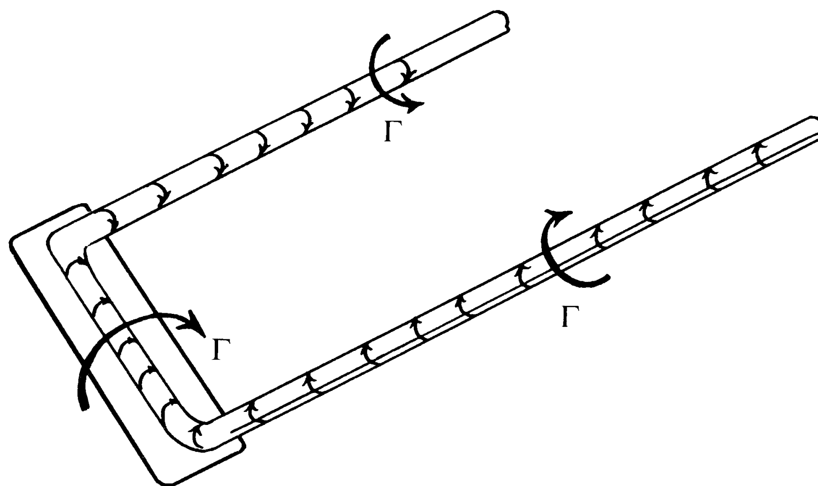


Figure 6-9: *The simplified horseshoe vortex*

Intensity of a vortex, i.e. total amount of vorticity passing through any plane region within a flow field is called *circulation* (Γ). From the Helmholtz's second theorem, the strength of the circulation around any section of the wing is the sum of the strengths of the vortex filaments cut by the section plane. As the section plane is progressively moved outwards from the centre section (root of the wing) to the tips, fewer and fewer bound vortex filaments are left for successive sections to cut so that the circulation around the sections diminishes. In this way, the spanwise change in circulation around the wing is related to the spanwise lengths of the bound vortices. Now, as the section plane is moved outwards along the bound bundle of filaments, and as the strength of the bundle decreases, the strength of the vortex filaments so far shed must increase, as the overall strength of the system cannot diminish. Thus the change in circulation from section to section is equal to the strength of the vorticity shed between these sections.

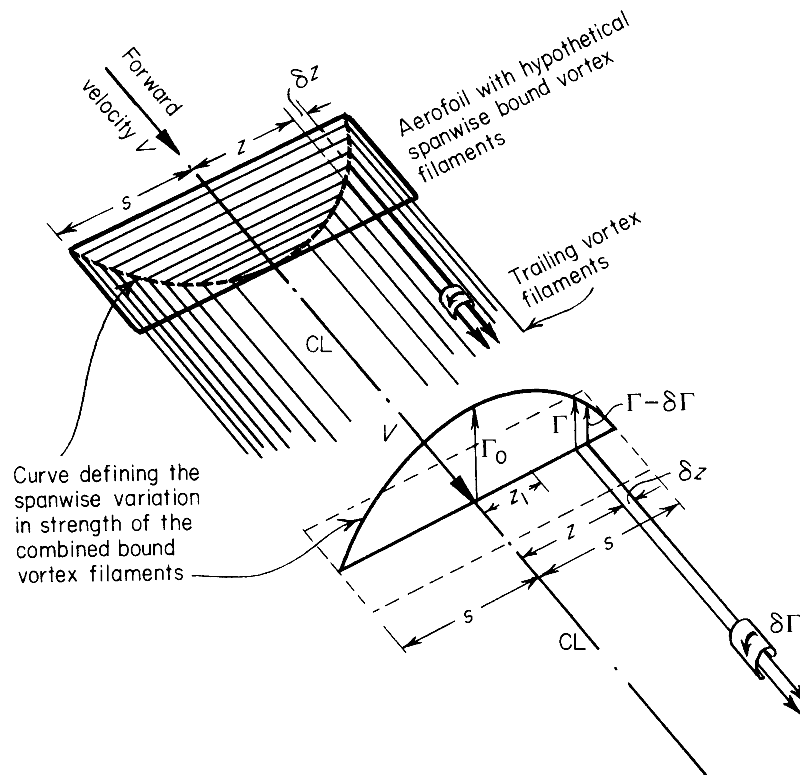


Figure 6-10: *The relation between spanwise load variation and trailing vortex strength*

Figure 6-10 shows a simple rectangular wing shedding a vortex trail with each pair of trailing vortex filaments completed by a spanwise bound vortex. A line joining the ends of all the spanwise vortices forms a curve that, assuming each vortex is of equal strength and give a suitable scale, would be a curve of the total strengths of the bound vortices at any section plotted against the span. This curve has been plotted for clarity on a spanwise line through the centre of pressure of the wing and is a plot of (chordwise) circulation Γ measured on a vertical ordinate, against spanwise distance

from the centre-line (CL) measured in the horizontal ordinate. Thus at a section z from the centre-line, sufficient hypothetical bound vortices are cut to produce a chordwise circulation around that section equal to Γ. At a further section $z+\delta z$ from the centre-line the circulation has fallen to $\Gamma - \delta\Gamma$, indicating that between section z and $z+\delta z$ trailing vorticity to the strength of $\delta\Gamma$ has been shed.

If the circulation curve can be described as some function of z, $f(z)$, then the strength of circulation shed is given by

$$\delta\Gamma = -df\frac{(z)}{dz}\delta z \quad .$$

Now at any section the lift per span is given by the Kutta-Zhukovsky theorem

$$l = \rho v \Gamma \tag{2}$$

and for given flight speed v and air density Γ is thus proportional to lift l. But l is the local intensity of lift or lift grading, which is either known or is the subject to the analysis.

The substitution of the wing by a system of bound vortices has not been rigorously justified at this stage. The idea allows a relation to be built up between the physical load distribution on the wing, which depends on the wing geometric and aerodynamic parameters and the trailing vortex system.

6.3.2 The elliptic distribution of lift

In order to demonstrate the general method of obtaining the aerodynamic characteristics of a wing from its loading distribution, the simplest load expression for symmetric flight is taken - a semi-ellipse. This substitution is held to be a good approximation to many (mathematically) more complicated distributions and is thus suitable for use as the first prediction in our performance estimates.

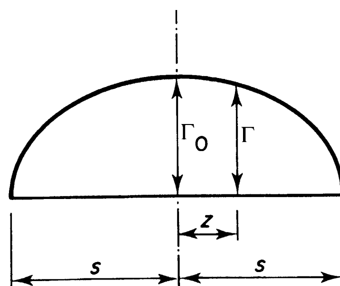


Figure 6-11: *Elliptic loading*

The spanwise variation in circulation is taken to be represented by a semi-ellipse having the span *2s* as the major axis and the circulation at mid-span Γ_0 as the semi-minor axis (see Figure 6-11). Then the general expression for an ellipse is

$$\frac{\Gamma^2}{\Gamma_0^2}+\frac{z^2}{s^2}=1 \quad \text{or} \quad \Gamma=\Gamma_0\sqrt{1-\left(\frac{z}{s}\right)^2} \quad . \tag{3}$$

Lift for elliptic distribution

By substituting (3) in

$$L=\int_s^s \rho\, v\, \Gamma\, dz$$

we get

$$L=\int_{-s}^{s} \rho\, c\, \Gamma_0\sqrt{1-\left(\frac{z}{s}\right)^2}\, dz=\rho\, v\, \Gamma_0 \pi \frac{s}{2},$$

whence

$$\Gamma_0=\frac{L}{\frac{1}{2}\rho\, v\, \pi\, s} \quad .$$

Or introducing

$$L=C_L \frac{1}{2}\rho\, v^2\, S$$

$$\Gamma_0=\frac{C_L\, v\, S}{\pi\, s}$$

giving the mid-span circulation in terms of the overall aerofoil lift coefficient C_L and geometry.

Downwash (induced velocity) for elliptic distribution

Derivation of equation for downwash results in

$$w=\frac{\Gamma_0}{4\mathrm{s}} \quad ,$$

which says that for a rectangular wing and elliptic distribution of lift is the downwash constant along the wing span.

Induced drag for elliptic distribution

From general equation for drag

$$D_V=\int_{-s}^{s} \rho\, w\, \Gamma\, dz \quad , \tag{4}$$

using the substitution of (3) in (4), induced drag for a rectangular wing and elliptic distribution of lift results in

$$C_{Dv}=\frac{C_L^2}{\pi\, AR} \quad , \tag{5}$$

where

$$aspect\ ratio\,(AR) = \frac{4s^2}{S} = \frac{span^2}{area} \quad .$$

Equation (5) establishes quantitatively how C_{Dv} falls with a rise in AR and also says that at zero lift in symmetric flight C_{Dv} is zero and the other condition is that as AR increases (to infinity for two-dimensional flow) C_{Dv} decreases (to zero).

6.3.3 The general distribution of lift

In the previous section, our attention was directed to the distribution of circulation (or lift) along the span, in which the load was assumed to fall symmetrically about the centre-line according to a particular family of load distributions. For steady symmetric manoeuvres this is quite satisfactory and the previous distribution formula may be arranged to suit certain cases. Its use, however, is strictly limited and it is necessary to seek further for an expression that will satisfy every possible combination of wing design parameter and flight manoeuvre. For example, it has so far been assumed that the wing was an isolated lifting surface that in straight steady flight had a load distribution rising steadily from zero at the tips to a maximum at mid-span (Figure 6-12a). The general wing, however, will have a fuselage located in the centre sections that will modify the loading in that region (Figure 6-12b) and engine nacelles or other excrescences may deform the remainder of the curve locally.

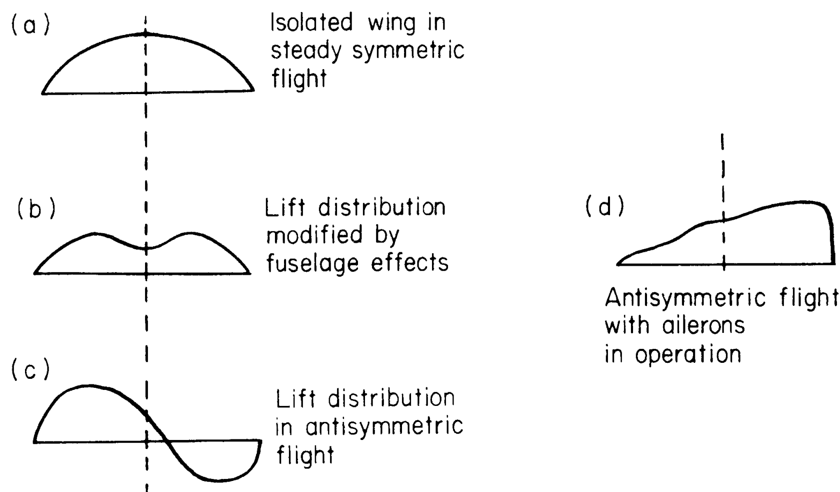


Figure 6-12: *Typical spanwise distribution of lift*

The load distributions on both the isolated wing and the general aeroplane wing will be considerably changed in anti-symmetric flight. In rolling, for instance, the upgoing wing suffers a large decrease in lift, which may become negative at some incidences (Figure 6-12c). With ailerons in operation, the curve of spanwise loading for a wing is no longer smooth and symmetrical but can be rugged and distorted in shape (Figure 6-12d). It is clearly necessary to have an expression that will

accommodate all these various possibilities. From previous work the formula $l=\rho v\Gamma$ for any section of span is familiar. Writing l in the form of the non-dimensional lift coefficient and equating to $\rho v\Gamma$

$$\Gamma = \frac{C_L}{2} vc$$

is obtained. This shows that for a given steady flight state the circulation at any section can be represented by the product of the forward velocity v and the local chord length c.

Now in addition the local chord can be expressed as a fraction of the semi-span s, and with this fraction absorbed in a new number and the numeral 4 introduced for later convenience, Γ becomes

$$\Gamma = 4 C_{\Gamma} s \quad ,$$

where C_{Γ} is dimensionless circulation, which will vary similarly to Γ across the span. In other words, C_{Γ} is the shape parameter or variation of the Γ curve and being dimensionless, it can be expressed as the Fourier series $\sum_1^{\infty} A_n \sin n\theta$, in which the coefficients A_n represent the amplitudes and the sum of the successive harmonics describes the shape. The sine series was chosen to satisfy the end conditions of the curve reducing to zero at the tips where $y=\pm s$. These correspond to the values of $\theta = 0$ and π. It is well understood that such a series is unlimited in angular measure but the portions beyond 0 and π can be disregarded here. Further, the series can fit any shape of curve but, in general, for rapidly changing distributions as shown by a rugged curve, for example, many harmonics are required to produce a sum that is a good representation.

In particular the series is simplified for the symmetrical loading case when the even terms disappear (Figure 6-13 II). For the symmetrical case, a maximum or minimum must appear at the mid-section. This is only possible for sines of odd values of $\pi/2$. That is, the symmetrical loading must be the sum of symmetrical harmonics. Odd harmonics are symmetrical. Even harmonics, on the other hand, return to zero again at $\pi/2$ where in addition there is always a change in sign. For any asymmetry in the loading one or more even harmonics are necessary.

With the number and magnitude of harmonics effectively giving all possibilities, the general spanwise loading can be expressed as

$$\Gamma = 4 s v \sum_1^{\infty} A_n \sin(n\theta).$$

It should be noted that since $l=\rho v\Gamma$ holds, the spanwise lift distribution can be expressed as

$$l = 4\rho v^2 s \sum_1^{\infty} A_n \sin(n\theta).$$

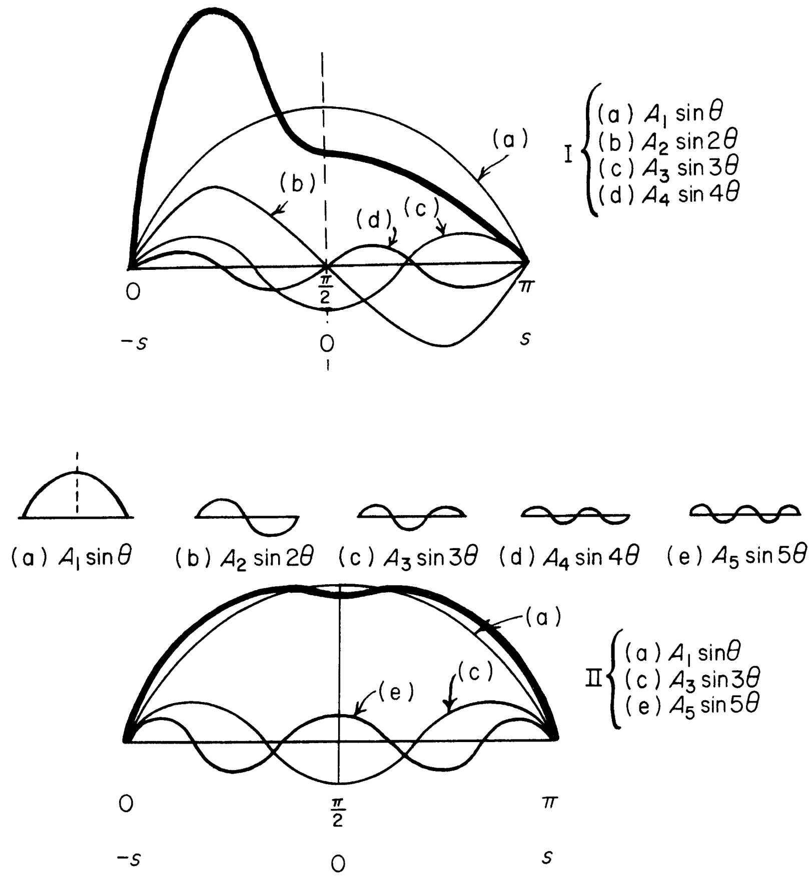


Figure 6-13: *Loading make-up by selected sine series*

6.3.4 Aerodynamic characteristics for symmetrical general loading

Lift on the wing

From the definition of lift

$$L = \int_{-s}^{s} \rho v \Gamma \, dz$$

by changing the variable $z = - s \cos\theta$

$$L = \int_0^{\pi} \rho v \Gamma s \sin\theta \, d\theta$$

and substituting for the general series expression, we get

$$\begin{aligned} L &= \int_0^{\pi} \rho v s^2 \sum A_n \sin n\theta \sin\theta \, d\theta \\ &= 4\mathrm{s}^2 \rho v^2 \int_0^{\pi} \sum A_n [\cos(n-1)\theta - \cos(n+1)\theta] \, d\theta \\ &= 4\mathrm{s}^2 \rho v^2 \frac{1}{2} \left[\sum A_n \left(\frac{\sin(n-1)\theta}{n-1} - \frac{\sin(n+1)\theta}{n+1} \right) \right]_0^{\pi}. \end{aligned}$$

The sum within the squared brackets equals zero for all values of $n \neq 1$ when it becomes

$$\left[\lim_{(n-1)\to 0} A_1 \frac{\sin(n-1)\theta}{(n-1)}\right]_0^{\pi} = A_1\pi \quad .$$

Thus

$$L = A_1 \pi \frac{1}{2} \rho v^2 4 s^2 = C_L \frac{1}{2} \rho v^2 S$$

and writing aspect ratio *(AR)* = $4s^2/S$ gives

$$C_L = \pi (AR) A_1 \quad .$$

This indicates the rather surprising result that the lift depends on the magnitude of the coefficient of the first term only, no matter how many terms may be present in the series describing the distribution. This is because the terms $A_3 \sin 3\theta$, $A_5 \sin 5\theta$, etc., provide positive lift on some sections and negative lift on others so that the overall effect of these is zero. These terms provide the characteristic variations in the spanwise distribution but do not affect the total lift of the whole, which is determined solely from the amplitude of the first harmonic. Thus

$$C_L = \pi (AR) A_1 \quad \text{and} \quad L = 2\pi \rho v^2 s^2 A_1 \quad .$$

Downwash

The derivation of equation for downwash results in

$$w = v \frac{\sum n A_n \sin n\theta}{\sin\theta} \quad .$$

This involves all the coefficients of the series and will be symmetrically distributed about the centre line for odd harmonics.

Induced drag

The resulting formula for induced drag is

$$C_{Dv} = \frac{C_L^2}{\pi (AR)} [1 + \delta] \quad ,$$

where

$$\delta = \left(\frac{3A_3^2}{A_1^2} + \frac{5A_5^2}{A_1^2} + \frac{7A_7^2}{A_1^2} + ... \right) \quad .$$

Plainly δ is always a positive quantity because it consists of squared terms that are always positive. C_{Dv} can be a minimum only if $\delta = 0$. That is if $A_3 = A_5 = A_7 = ... = 0$ and the only term remaining

in the series is $A_1 sin\theta$.

6.3.5 Determination of the load distribution on a wing

Determination of the load distribution on a wing is the direct problem broadly facing designers who wish to predict the performance of a projected wing before the long and costly process of model tests begins. This does not imply that such tests need not to be carried out. On the contrary, they may be important steps in the design process towards a production aircraft.

The problem can be rephrased as follows. The designers wish to have some indication how the wing characteristics vary as, for example, change of the geometric parameters of the projected wing. In this way they can balance the aerodynamic effects of their changing ideas against the basic specification – provided there is a fairly simple process relating the changes in design parameters to the aerodynamic characteristics. Of course, this is stating one of the design problems in its baldest and simplest terms, but as in any design work, plausible theoretical processes yielding reliable predictions are very comforting.

The general theory for wings of high aspect ratio

A start is made by considering the influence of the end effect, or downwash, on the lifting properties of an aerofoil section of some distance z from the centre-line of the wing. Figure 6-14 shows the lift-versus-incidence curve for an aerofoil section of a certain profile working two-dimensionally and working in flow regime influenced by end effects, i.e. working at some point along the span of a finite lifting wing.

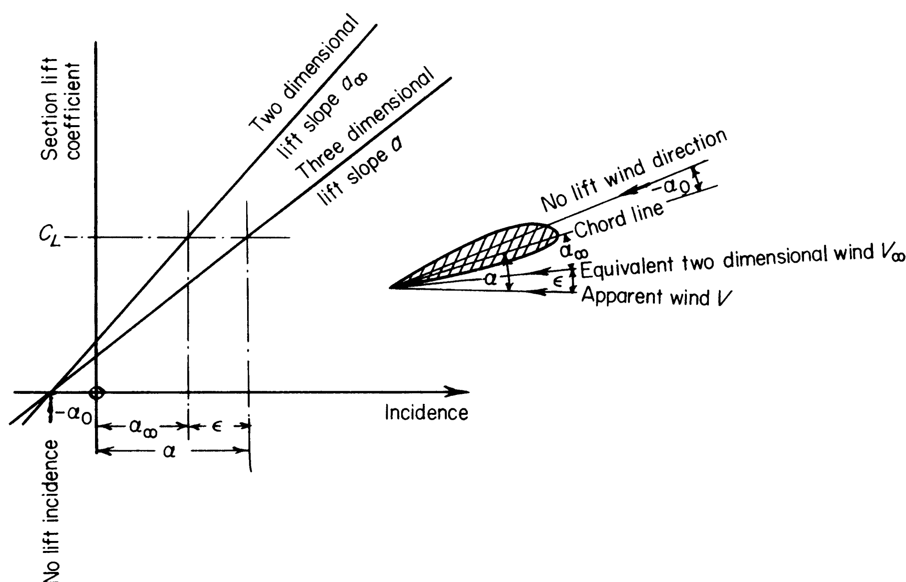


Figure 6-14: *Lift-versus-incidence characteristic for an aerofoil section of a certain profile*

Assuming that both curves are linear over the range considered, i.e. the working range, and that

under both flow regimes the zero-lift incidence is the same, then

$$C_L = a_\infty[\alpha_\infty - \alpha_0] = a[\alpha - \alpha_0] \quad .$$

Taking the first equation with $\alpha_\infty = \alpha - \varepsilon$, we have

$$C_L = a_\infty[(\alpha - \alpha_0) - \varepsilon] \quad . \tag{6}$$

But at the same time from (2)

$$C_L = \frac{lift\ per\ unit\ span}{\frac{1}{2}\rho v^2 c} = \frac{l}{\frac{1}{2}\rho v^2 c} = \frac{\rho v \Gamma}{\frac{1}{2}\rho v^2 c}, \qquad C_L = \frac{2\Gamma}{vc}. \tag{7}$$

By equating (6) and (7) and rearranging, we get

$$\frac{2\Gamma}{ca_\infty} = v[(\alpha - \alpha_0) - \varepsilon] \quad .$$

And since

$$v\varepsilon = w = -\frac{1}{4\pi}\frac{\int_{-s}^{s} d\Gamma/dz}{z - z_1} dz \quad ,$$

we get

$$\frac{2\Gamma(z)}{c(z)a_\infty} = v(\alpha - \alpha_0) + \frac{1}{4\pi}\int_{-s}^{s}\frac{(d\Gamma/dz)}{z - z_1} dz \tag{8}$$

Equation (8) is the Prandtl's integral equation for the circulation Γ at any section along the span in terms of all the aerofoil parameters. The solution of this equation cannot be found analytically for all points along the span but only numerically at selected spanwise stations and at each end of the wing.

Glauert's solution to Prandtl's equation

This method is based on utilisation of Fourier series to solve the equation (8). We transform the *z* variable as follows:

$$z = -\frac{l}{2}\cos\theta$$

and denote *l/2* = *s* (semi-span). Since

$$z_1 = -\frac{l}{2}\cos\theta_1 \quad ; \quad dz_1 = \frac{l}{2}\sin\theta_1 \, d\theta_1 \quad ; \quad \frac{d\theta_1}{dz_1} = \frac{2}{l\sin\theta_1} \quad ,$$

thus

$$\begin{gathered} z_1 - z = -\frac{l}{2}(\cos\theta_1 - \cos\theta), \\ \frac{d\Gamma}{dz_1} = \frac{d\Gamma}{d\theta_1}\frac{d\theta_1}{dz_1} = \frac{2}{l}\frac{d\Gamma}{d\theta_1}\frac{1}{\sin\theta_1}. \end{gathered} \tag{9}$$

And after substituting (9) into (8) we get

$$\Gamma = \frac{1}{2} a_\infty v_\infty c \left[\alpha_a - \frac{1}{2\pi l v_\infty} \int_0^\pi \frac{\frac{d\Gamma}{d\theta_1} d\theta_1}{\cos\theta_1 - \cos\theta} \right] \quad . \tag{10}$$

Because $\Gamma = 0$ for $\theta = 0$ and $\theta = \pi$ and Γ reaches maximum values for $\theta = \pi/2$, the Fourier series contains only $\sin n\theta_1$ terms. Then the circulation is

$$\Gamma = 2 l v_\infty \sum_{n=1}^{\infty} A_n \sin n\theta_1 \quad , \tag{11}$$

where the unknowns A_n are to be found. By differentiating (11) we obtain

$$\frac{d\Gamma}{d\theta_1} = 2 l v_\infty \sum_{n=1}^{\infty} n A_n \cos n\theta_1 \quad . \tag{12}$$

Equating (12) to (10)

$$2 l v_\infty \sum_{n=1}^{\infty} A_n \sin n\theta_1 = \frac{1}{2} a_\infty v_\infty c \left[\alpha_a - \frac{1}{\pi} \int_0^\pi \frac{\sum_{n=1}^{\infty} n A_n \cos n\theta_1}{\cos\theta_1 - \cos\theta} d\theta_1 \right] \quad . \tag{13}$$

Because nA_n is for each term of the series a constant, we can express (13) as

$$2 l \sum_{n=1}^{\infty} A_n \sin n\theta_1 = \frac{1}{2} a_\infty c \left[\alpha_a - \frac{1}{\pi} \sum_{n=1}^{\infty} n A_n \int_0^\pi \frac{\cos n\theta_1}{\cos\theta_1 - \cos\theta} d\theta_1 \right] \quad . \tag{14}$$

Since

$$\int_0^\pi \frac{\cos n\theta_1}{\cos\theta_1 - \cos\theta} d\theta_1 = \pi \frac{\sin n\theta}{\sin\theta}$$

the equation (14) after rearrangement read as

$$a_\infty c \sum_{n=1}^{\infty} A_n n \sin n\theta + 4l \sin\theta \sum_{n=1}^{\infty} A_n \sin n\theta = a_\infty \alpha_a x \sin\theta \quad .$$

By substituting

$$\frac{a_\infty c}{4l} = \mu$$

we obtain

$$\sum_{n=1}^{\infty} (\mu n + \sin\theta) \sin n\theta \cdot A_n = \mu \alpha \sin\theta \quad .$$

This equation is applicable for all stations on the semi-span, using it we can calculate the values A_n. For symmetrical wing and symmetric flow, it is also the distribution of circulation symmetric, thus

$$\Gamma(\theta) = 2 l v_\infty \sum_{n=1}^{\infty} A_n \sin n\theta = 2l v_\infty \sum_{n=1}^{\infty} A_n \sin n(\pi - \theta) = \Gamma(\pi - \theta).$$

Because for even n is $\sin n(\pi-\theta) = -\sin n\theta$ and for odd n is $\sin n(\pi-\theta) = \sin n\theta$, it is clear that coefficients A_n with even n are equal to zero. They are not equal to zero only for non-symmetrical wing (with ailerons in operation) or for non-symmetrical flow.

6.3.6 Optimised wing model

Aerodynamic model of the wing is based on the Glauert's solution of Prandtl's equation (8) described above. We do not use linear lift slope characteristics for computing local lift coefficient. They were replaced by full non-linear aerofoil lift and drag characteristics to provide more accurate results, especially in cases close to critical angle of attack.

The wing was divided into three sections (see Figure 6-15) to enable us to manipulate with the following geometric parameters of the wing:

1. Section intermediate point 1 – y coordinate of transition between sections 1 and 2,
2. Section intermediate point 2 – y coordinate of transition between sections 2 and 3,
3. Chord length 1 – length of the root chord of the wing (profile cut next to the fuselage),
4. Chord length 2 – length of the chord on the transition between sections 1 and 2,
5. Chord length 3 – length of the chord on the transition between sections 2 and 3,
6. Chord length 4 – length of the tip chord,
7. Twist 1 – geometric twist of the root chord,
8. Twist 2 – geometric twist of the chord between sections 1 and 2,

9. Twist 3 – geometric twist of the chord between sections 2 and 3,

10. Twist 4 – geometric twist of the tip chord,

11. Wing span

12. Profile 1 – aerofoil type for the root cut,

13. Profile 2 – aerofoil type for cut between sections 1 and 2,

14. Profile 3 – aerofoil type for the cut between sections 2 and 3,

15. Profile 4 – aerofoil type for the tip cut.

The ranges of optimised parameters were set according to the requirements for each particular aeroplane. Coordinates of section intermediate points were usually set between 1m and the wing semi-span, chord lengths limited between 0.1 and 2 m (however, the root chord length was mostly defined by the dimensions of the centroplane), geometric twist of the profiles represents absolute rotation of the particular cut against plane fuselage. Wing span range is given by the aeroplane loading, for example for ultralights was set between 7 and 12 meters. There were two available models of aerofoils for parameters 12-14 – LS0417MOD and MS0313.

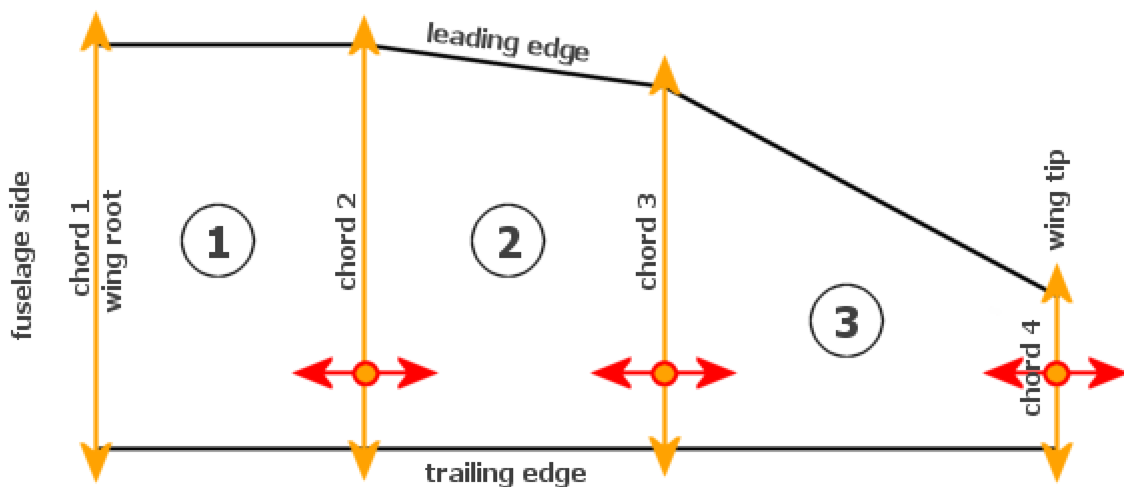


Figure 6-15: *Modified parameters of a wing*

The following computed output parameters were to be minimised:

1. induced drag,

2. surface-friction drag,

3. overall wing area,

4. difference $C_L – 0.9*C_{LMax}$ in 70% of semi-span,
5. difference $C_{LMax} – C_L$ in y = <0, 40%> of semi-span.

Last two items represent requirements on shape of the lift curve. To meet the requirements on wing stall characteristics, there must be at least 10% reserve of lift on the ailerons when the flow on root part of the wing is starting to separate (air flow separates from the wing at high angles of attack or low speeds; the value of critical angle of attack and minimum speed is given by used aerofoils). The second requirement maximises the overall lift of the wing.

The resulting computation of fitness is represented by weighted sum of minimised parameters:

$$\begin{aligned} costValue &= inducedDrag \cdot 100 + frictionDrag \cdot 100 + S/10 \\ &+ 40\mathrm{pDiff} \cdot 100 + 70\mathrm{pDiff} \cdot 100 + penalisation \end{aligned} .$$

There is one additional requirement on the wing expressed by the *penalisation* value. It encapsulates the condition

$$S \cdot C_{LMax}(wing) \geqslant k \quad , \quad k = \frac{2 \cdot m \cdot g}{\rho \cdot v^2} \quad ,$$

where S is the wing surface, C_{LMax} the maximal lift of the wing, m stands for weight of the airplane, g = 9.81, ρ represents air density and v denotes velocity. It is desired to keep $S * C_{LMax}$ as close as possible to k but not below.

The stall properties of the wing (requirements on shape of the lift curve) are computed at *stall speed* given by FAA directives (here 45 knots) and the other properties at maximal speed of steady level flight (given by construction of particular aeroplane).

6.3.7 Optimisation results

In this section, wing optimisation results for different aeroplanes are presented. The first one is a proposal of modification of the wing for the SportStar ultralight plane. The second group of optimisation was performed in order to modify the VUT-100 Cobra wing to improve its properties. The third group of evolved wings is a study of a completely new wing for VUT-100 Cobra without any restrictions used in the previous simulations.

Figure 6-16: *Aeroplanes from Evektor production: a two-seated very light aeroplane "Harmony" (left) and the four-seated VUT100 Cobra (right). Source: www.evektor.cz*

SportStar

Wing optimisation of this ultralight aeroplane was performed under following constraints:

- wing consists of 2 sections (2 and 3; length of section 1 set to zero),
- length of chords 1, 2 and 3 is fixed at 1.25 m (geometry section 1 is not optimised),
- maximum wing-span is 12 m,
- minimum *y* coordinate of chord 3 (transition between sections 2 and 3) is 3.5 m,
- twist of chords 1, 2 and 3 is zero, twist of chord 4 allowed between -5 and +5 degrees,
- LS0417MOD aerofoils are used for chords 1, 2 and 3, type of aerofoil for chord 4 is chosen by SOMA,
- $S * C_{LMax} > 18.4$,
- stall speed 45 knots, maximum speed 108 knots.

The resulting wing parameters are summarised in Table 6-2 and the values obtained from the optimisation process can be seen in Table 6-3.

Wing area =	11.7942	Wing C_L =	0.3476
C_{LMax} =	1.6236	Friction drag =	0.0108
$S*C_{LMax}$ =	19.1492	Induced drag =	0.0250

Table 6-2: *SportStar: wing properties*

Section intermediate point 1 = 0	Twist 1 = 0
Section intermediate point 2 = 3.5000	Twist 2 = 0
Chord length 1 = 1.2500	Twist 3 = 0
Chord length 2 = 1.2500	Twist 4 = -4.9999
Chord length 3 = 1.2500	Profile 1 = LS0417MOD
Chord length 4 = 0.1000	Profile 2 = LS0417MOD
Wing span = 11.5100	Profile 3 = LS0417MOD
	Profile 4 = MS0313

Table 6-3: *SportStar: overview of optimised wing parameters*

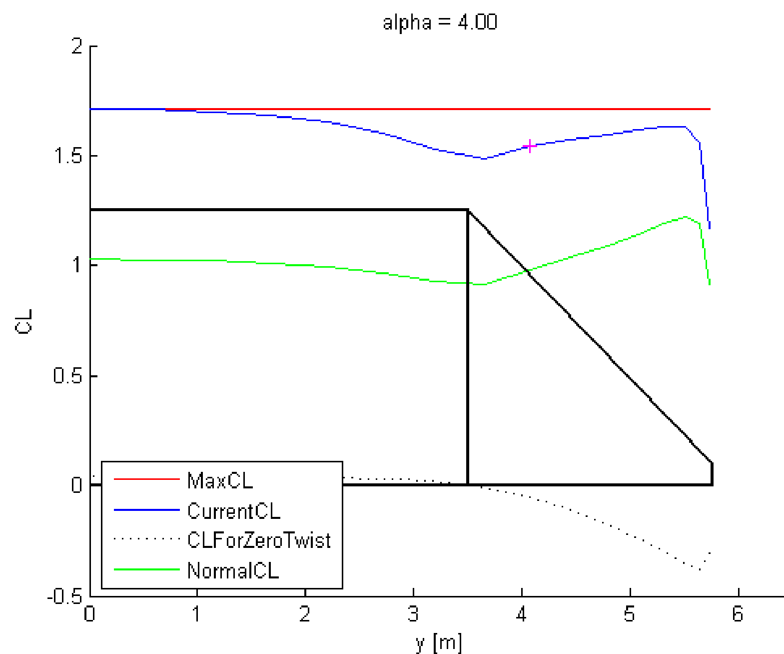


Figure 6-17: *SportStar: shape of evolved wing and its lift characteristics*

Let us explain the lift diagrams: MaxCL (red) stands for maximum lift of the wing. This value is given by the maximum lift of used aerofoils. The CurrentCL (blue) represents current state of lift distribution along the span. By increasing the angle of attack this curve changes its shape according to actual state. CLForZeroTwist describes spanwise lift for the case the wing is under such an angle of attack that it has zero lift. This curve shows the influence of geometric twist on lift of the wing. And finally, the NormalCL curve (green) symbolises lift normalised to 1 at the root profile. By multiplying this value on selected station along the span by local lift coefficient, we get value of current lift. Small cross (magenta) in 70% of the wing semi-span indicates the 10% reserve of lift on the ailerons. Curve of the current CL must go through this point to ensure good wing stall characteristics of the wing.

VUT-100 Cobra

VUT-100 Cobra is an all-metal four-seat aircraft comming to serial production in 2006. We decided to compare the already designed wing with a wing optimised by SOMA. The constraint conditions of the wing were as described below:

- wing consists of 3 sections,
- length of chords 1, 2 fixed at 1.579 m (centroplane section),
- maximum wing-span 12 m,
- y coordinate of chord 2 fixed at 1.25 m (centroplane section length)
- y coordinate of chord 3 (transition between sections 2 and 3) may vary between 3 m and the wing-span,
- twist of chords 1 fixed to zero, twist of chord 1 allowed between -1 and 1 degree, twist of chord 3 allowed from -3 to +3 degrees and twist of chord 4 (tip) allowed between -5 and +5 degrees,
- LS0417MOD aerofoils are used for chords 1 and 2, type of aerofoils for chords 3 and 4 are optimised,
- $S * C_{LMax} > 20.8$,
- stall speed 45 knots, maximum speed 120 knots.

Values obtained from the optimisation process are:

Section intermediate point 1 = 1.2765	Twist 1 = 0
Section intermediate point 2 = 3.0046	Twist 2 = 0.9034
Chord length 1 = 1.5790	Twist 3 = -3.7972
Chord length 2 = 1.5790	Twist 4 = -5.6718
Chord length 3 = 1.2001	Profile 1 = LS0417MOD
Chord length 4 = 0.2933	Profile 2 = LS0417MOD
Wing span = 11.9849	Profile 3 = MS0313
	Profile 4 = MS0313

Table 6-4: *VUT-100 Cobra, approach I: overview of optimised wing parameters*

Wing area = 13.2957	Wing C_L = 0.1939
C_{LMax} = 1.6121	Friction drag = 0.0098
$S*C_{LMax}$ = 21.4344	Induced drag = 0.0085

Table 6-5: *VUT-100 Cobra, approach I: wing properties*

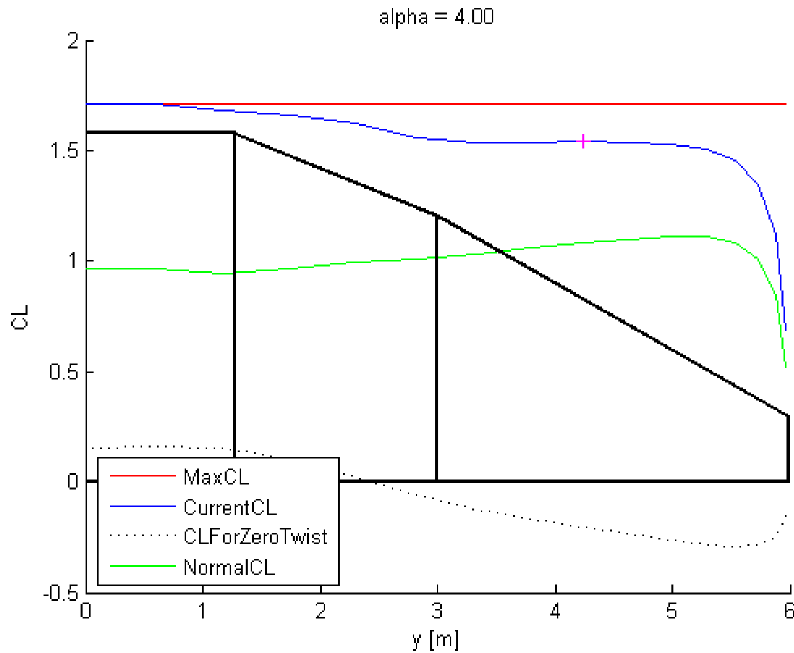


Figure 6-18: *VUT-100 Cobra, approach I: shape of evolved wing and its lift characteristics*

Geometry of this wing seems to be very acceptable, nevertheless, there is one substantial problem. There might be doubts about construction of the main girder as the sweep-back angle and the wing-depth reduction from root to tip is rather significant. Therefore, we decided to modify the constraints – to fix the length of chord 3 also at 1.579 meters. Below you can assess results of this modification:

Section intermediate point 1 = 1.2765	Twist 1 = 0
Section intermediate point 2 = 3.0000	Twist 2 = 0
Chord length 1 = 1.5790	Twist 3 = 0
Chord length 2 = 1.5790	Twist 4 = -5.0000
Chord length 3 = 1.5790	Profile 1 = LS0417MOD
Chord length 4 = 0.5101	Profile 2 = LS0417MOD
Wing span = 12.0000	Profile 3 = LS0417MOD
	Profile 4 = LS0417MOD

Table 6-6: *VUT-100 Cobra, approach II: overview of optimised wing parameters*

Wing area = 15.7412	Wing C_L = 0.3048
C_{LMax} = 1.5976	Friction drag = 0.0100
$S*C_{LMax}$ = 25.1490	Induced drag = 0.0290

Table 6-7: *VUT-100 Cobra, approach II: wing properties*

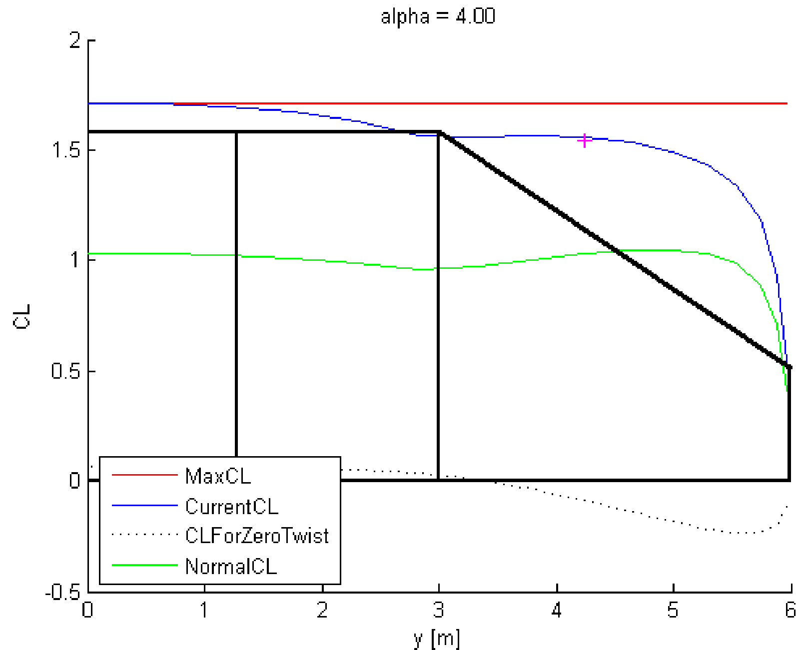


Figure 6-19: *VUT-100 Cobra, approach II: shape of evolved wing and its lift characteristics*

From the construction point of view is this shape of the wing much more feasible than the previous one. The increase in the wing area also increased the overall lift, however, also the values of both drags are higher. There is only a very narrow space for the optimisation algorithm to reach the desired wing parameters. You can also see that the $S*C_{LMax}$ parameter is quite far from the requested value and high twist on chord 4 tries to improve the wing characteristics in a very similar way as it happened also in the SportStar's wing optimisation case.

A very interesting question arises here: What would be the best wing geometry of the VUT-100 Cobra aeroplane? To find a satisfactory answer we must remove some of the expendable constraints.

Unconstrained parameters for VUT-100 Cobra wing

In this approach we wanted to find the best possible wing for the VUT-100 Cobra aeroplane. We omitted all constraints except the loading capacity requirement – $S * C_{LMax} > 20.8$.

The resulting wing we obtained proves superior performance – very low drag and lift characteristics very close to the desired values. However, the aerofoil of chord 1 (MS0313) is not usually used for root profile of the wing for construction reasons (not matter of aerodynamics).

Section intermediate point 1 = 0.2735	Twist 1 = 0
Section intermediate point 2 = 1.1849	Twist 2 = -1.9775
Chord length 1 = 1.4668	Twist 3 = -2.4952
Chord length 2 = 1.4465	Twist 4 = -5.8831
Chord length 3 = 1.3099	Profile 1 = MS0313
Chord length 4 = 0.7758	Profile 2 = LS0417MOD
Wing span = 11.8136	Profile 3 = LS0417MOD
	Profile 4 = LS0417MOD

Table 6-8: *VUT-100 Cobra, approach III: overview of optimised wing parameters*

Wing area = 13.1574	Wing C_L = 0.0568
C_{LMax} = 1.5827	Friction drag = 0.0099
$S*C_{LMax}$ = 20.8238	Induced drag = 0.0008

Table 6-9: *VUT-100 Cobra, approach III: wing properties*

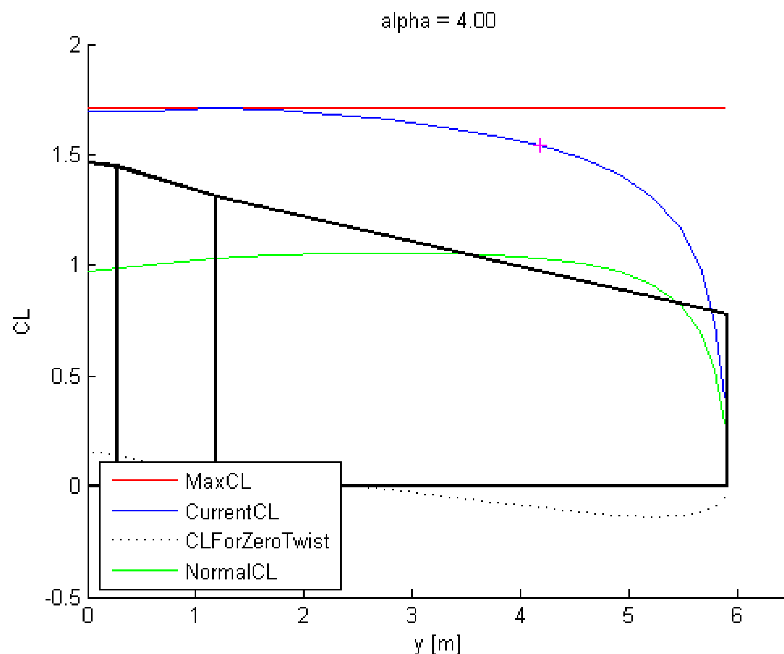


Figure 6-20: *VUT-100 Cobra, approach III: shape of evolved wing and its lift characteristics*

Note on the chord #3 y-coordinate limitation

The optimisation process tends to evolve wings with a high aspect ratio. This trend is mainly supported by the fact, that a wing with these characteristics has significantly lower values of drag (from the theory of aerodynamics infinite wings have zero drag). By demanding minimisation of the wing area (the $S*C_{LMax}$ condition) and drag we force SOMA to make wings more slender and longer.

Section intermediate point 1 = 1.2500	Twist 1 = 0
Section intermediate point 2 = 1.5716	Twist 2 = 0.8752
Chord length 1 = 1.5790	Twist 3 = -2.9948
Chord length 2 = 1.5790	Twist 4 = -4.8837
Chord length 3 = 1.3285	Profile 1 = LS0417MOD
Chord length 4 = 0.6889	Profile 2 = LS0417MOD
Wing span = 11.2948	Profile 3 = MS0313
	Profile 4 = LS0417MOD

Table 6-10: *VUT-100 Cobra, the dead end: overview of optimised wing parameters*

Wing area = 13.1048	Wing C_L = 0.1548
C_{LMax} = 1.5869	Friction drag = 0.0097
$S*C_{LMax}$ = 20.7968	Induced drag = 0.0067

Table 6-11: *VUT-100 Cobra, the dead end, approach III: wing properties*

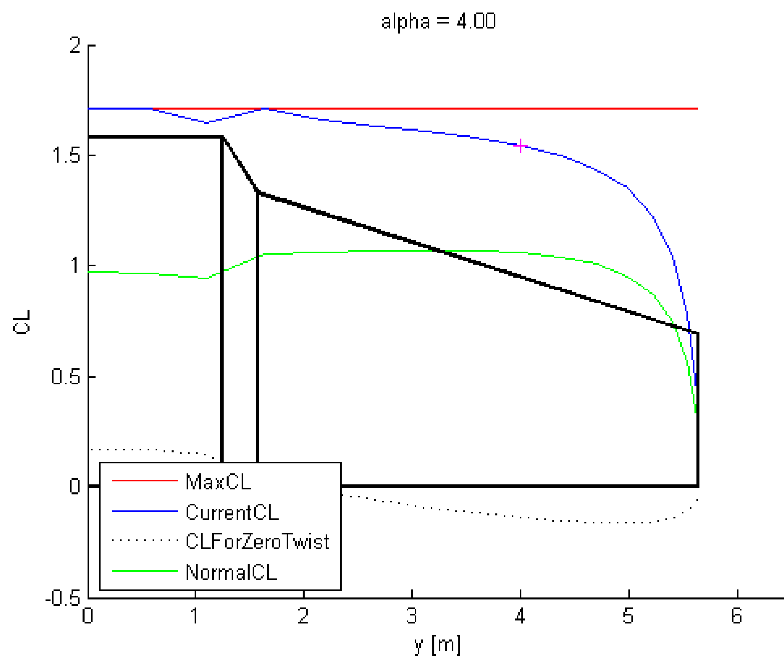


Figure 6-21: *VUT-100 Cobra, the dead end: shape of evolved wing and its lift characteristics*

But there is still the question why we limited the *y*-coordinate of chord #3 to 3 metres. If we had not, we would obtain a wing geometry as displayed in Figure 6-21. Considering the wing parameters, we obtained a high performance wing (compare to VUT-100 Cobra approach I wing parameters in Figure 6-18, tables 6-6 and 6-7), however, the construction of a wing of this shape might be a bit tricky. By setting the length of section 2 to a fixed value, we limit the sweep-back angle of the leading edge to reasonable values.

The second reason why, without this chord 3 *y*-coordinate constraint, we obtain such shapes of wings is that in the model there are included only aerodynamic characteristics without any construction limitations.

6.4 Chapter Summary

This chapter entirely describes three applications of parallel SOMA, which were successfully accomplished in cooperation with the Strathclyde University in Glasgow, the Helsinki University of Technology and the Evektor company.

In the first assignment, parameters of a modern four cylinder internal combustion engine were a subject to optimisation. The output parameters – fuel consumption, amount of exhaust and deviation from the desired torque – were minimised by optimising four input parameters – manifold pressure, inlet valve timing, exhaust valve timing and spark advance for given combinations of engine's setpoints - RPM (revolutions per minute) and torque. Created software and simulation results obtained from computer model of the engine was later used by a North-American car producer for parametric optimisation directly on a real engine.

The second challenging task for parallel SOMA was the optimisation of relay nodes positions aggregating and forwarding data from wireless sensor array to a central data-collection node. The aim of this work was to develop a method for finding optimal positions for a certain number of relay nodes to maximise the lifetime of a battery-powered network. Comparing to conventional algorithms, SOMA demonstrated an alternative approach to a complex engineering problem.

The third and the most impressive application was the aerodynamic optimisation of wing geometry for an aeroplane being prepared for production by a Czech leading aircraft producer, the Evektor company. Obtained results meet the desired wing parameters and support assumptions made by experts on aerodynamics. Wings evolved by SOMA tend to be of high aspect ratio with high values of lift and low drag. Created aerodynamic model together with developed optimisation software will be used as a requisite for future wing design in the company.

Described models of combustion engine and the wing were optimised by parallel SOMA using the UDPOptCluster extension due to their implementation in the Matlab environment (UDPOptCluster represents a message passing interface (MPI), especially developed for parallelising SOMA on the MathWork's Matlab and Wolfram's Mathematica platforms). The relay node optimisation was performed using the Cluster framework as parts of the model were a combination of C++ and Java code.

7 CONCLUSIONS

Evolutionary algorithms based on principles of natural selection belong to very efficient methods of global optimisation. They use mechanisms inspired by biological evolution: reproduction, mutation, recombination, natural selection and survival of the fittest. Evolutionary algorithms perform consistently well approximating solutions to all types of problems and are able to find a feasible solution of many engineering problems in a reasonable time. However, when employing them for more complex problems, the time required for finding a suitable solution might be unacceptably long. Therefore, many researchers are concerned with parallelisation of various genetic and evolutionary algorithms. By employing computational clusters, grids, parallel computers and networks of workstations we can broaden their sphere of activity on complicated heavy-computational optimisation tasks.

This work is primarily aimed to parallel optimisation evolutionary techniques. The main contribution of this thesis lies in the analysis, implementation and empirical validation of parallel SOMA evolutionary algorithm.

At the beginning of this work, the current state on the field of parallel genetic and evolutionary algorithms is summarised. This overview starts with classification of parallel genetic algorithms, summarizes the history and characterises various parallelisation approaches and communication strategies in detail.

The first essential step to parallelise the SOMA algorithm was to prepare a foundation, on which the further work stands. As the foundation stone, a fully scalable, high-performance, universal and multiplatform framework for parallel/distributed applications was developed. This versatile cluster platform was designed in order to utilize free CPU time of common office computers that are present in relatively high numbers at all universities and many companies and are used for ordinary non-demanding office tasks. It proved very decent performance and high reliability during all performed tests. With efficiency about 99 % is the framework ready to be used anywhere where there is a need for high-performance computational tasks.

After considering numerous parallelisation models and analysing parallel implementations of various evolutionary algorithms, we chose those promising high efficiency of the complete distributed system. Since our parallelised application (SOMA) does not require synchronism for its run, we were able to avoid this factor, which may significantly decrease the overall performance.

A separate chapter is devoted to an analysis of parallelisation models of Differential Evolution (DE) and its already existing parallel implementations. Consequently, a concept of parallel SOMA

inspired by parallel DE approaches is presented and thoroughly described. The following parallelisation strategies were chosen to parallelise SOMA (in order from less to most efficient): synchronous island model, asynchronous island model I (using the cluster framework), asynchronous island model II (running in the UDPOptCluster) and cellular/diffusion model using the cluster's Virtual network feature. The last model appears to be the most suitable for parallelising the SOMA algorithm and brings the highest optimisation performance. In several studies of parallel DE, some improvements in robustness of the algorithm are described, however, search for any algorithmic improvements of SOMA brought by parallelisation was not successful.

To validate algorithmic qualities and high optimisation performance of parallel SOMA, three real-world engineering applications were successfully accomplished in cooperation with the Strathclyde University in Glasgow, the Helsinki University of Technology and the Evektor company.

In the first assignment, parameters of a modern four cylinder internal combustion engine were a subject to optimisation. The output parameters – fuel consumption, amount of exhaust and deviation from the desired torque – were minimised by optimising four input parameters – manifold pressure, inlet valve timing, exhaust valve timing and spark advance for given combinations of engine's setpoints - RPM (revolutions per minute) and torque. Created software and simulation results obtained from the computer model of the engine was later used by a North-American car producer for parametric optimisation directly on a real engine.

The second challenging task for parallel SOMA was the optimisation of relay nodes positions aggregating and forwarding data from wireless sensor array to a central data-collection node. The aim of this work was to develop a method for finding optimal positions for a certain number of relay nodes to maximise the lifetime of a battery-powered network. Comparing to conventional algorithms, SOMA reached very similar results and demonstrated an alternative approach to a complex engineering problem.

The third and the most impressive application was aerodynamic optimisation of wing geometry for an aeroplane being prepared for production in the Evektor company, a leading civil aircraft producer in the Czech Republic. There were 15 optimised parameters minimising induced drag, surface-friction drag and overall wing area. Furthermore, to meet the directives on wing stall characteristics, there was an requirement on shape of the lift curve. Results obtained for various wing configurations meet the desired wing parameters and support assumptions made by experts on aerodynamics. Wings evolved by SOMA tend to be of high aspect ratio with high values of lift and low drag. Created aerodynamic model together with developed optimisation software will be

used as a requisite for future wing design in the company.

To sum up, this work describes the current state on the field of parallel genetic/evolutionary algorithms and analyses parallelisation principles of various algorithms with the focus on Differential Evolution. Then, a parallelisation schema of the SOMA algorithm is proposed and four different implementations carefully benchmarked. In the last part of this work, three very successful applications of optimisation using SOMA evolutionary algorithm were presented. All of them demonstrate pioneering approach to engineering problems to which companies face every day. As the World is full of heavy and complex optimisation tasks, there will always be countless number of problems which SOMA can help to deal with. An infinity number of landscapes in which the SOMA's wolves can look for better and richer source of food.

REFERENCES

D. Abramson, J. Abela. *A parallel genetic algorithm for solving the school timetabling problem*. Proceedings of the Fifteenth Australian Computer Science Conference (ACSC-15), vol. 14, p. 1–11, 1992.

P. Adamidis. *Parallel Evolutionary Algorithms: A Review.* 4th Hellenic-European Conference on Computer Mathematics and its Applications. 1998.

E. Alba, J. M. Troya. *Analyzing synchronous and asynchronous parallel distributed genetic algorithms*. Generation Computer Systems, 17(4):451-465, 2001.

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

E. Alba, A. J. Nebro, J. M. Troya. *Heterogeneous Computing and Parallel Genetic Algorithms*. Journal of Parallel and Distributed Computing, 62(9):1362-1385, 2002.

M. Arenas, P. Collet, A. Eiben, M. Jelasity, J. Merelo, B. Paechter, M. Preuß M. Schoenauer. *A Framework for Distributed Evolutionary Algorithms*. Parallel Problem Solving from Nature VII, Granada, Spain, 665-675, 2002.

T. C. Belding. *The distributed genetic algorithm revisited*. Proceedings of the Sixth International Conference on Genetic Algorithms, San Francisco, CA, 114-121, 1995.

A. D. Bethke. *Comparison of genetic algorithms and gradient-based optimizers on parallel processors: Efficiency of use of processing capacity.* Tech. Rep. No. 197, Logic of Computers Group, University of Michigan, 1976.

A. Bevilacqua, R. Campanini, N. Lanconelli. *A Distributed Genetic Algorithm for Parameters Optimization to Detect Microcalcifications in Digital Mammograms*. Applications of Evolutionary Computing EvoWorkshops:EVOIASP, Como, Italy, 278-287, 2001.

M. Bhardwaj, A. Chandrakasan. B*ounding the Lifetime of Sensor Networks via Optimal Role Assignment*. Proc. IEEE Intl. Conf. On Communtications, Vol. 3, p. 785-790, 2002.

C. H. Braun. *On Solving Travelling Salesman Problems by Genetic Algorithms*. Parallel Problem Solving from Nature, p. 129–133, Springer-Verlag (Berlin), 1990.

E. Cantú-Paz. *Efficient and Accurate Parallel Genetic Algorithms*. Kluwer Academic Publishers, 162, 2000.

E. Cantú-Paz, M. Mejía-Olivera. *Experimental Results in Distributed Genetic Algorithms*. In International Symposium on Applied Corporate Computing, p. 99–108, Monterrey, Mexico, 1994.

E. Cantú-Paz, D. E. Goldberg. *Modeling Idealized Bounding Cases of Parallel Genetic Algorithms*. Genetic Programming 1997 : Proceedings of the Second Annual Conference, Morgan Kaufmann (San Francisco, CA), 1997.

M. Červenka, I. Zelinka. Relay Node Placement in Energy-*Constrained Networks Using SOMA Evolutionary Algorithm*. IASTED International Conference on Artificial Intelligence and Applications (AIA2006), Innsbruck, Austria, 2006, ISBN 0-88986-558-2.

M. Chu, H. Haussecker, F. Zhao. *Scalable Information-driven Sensor Querying and Routing for Ad-hoc Heterogenous Sensor Networks*. Intl. journal on High Performance Computing Applications 16, p. 293-313, 2002.

J. P. Cohoon, S. U. Hedge, W. N. Martin, D. Richards. *Punctuated equilibria: a parallel genetic algorithm*. Proc. Second Int. Conf. On Genetic Algorithms, Pittsburg, PA, 148-154, 1987.

V. Cung, L. S. Martins, C. C. Ribeiro, C. Roucairol. *Strategies for the Parallel Implementation of Metaheuristics*. Laboratoire PRiSM-CNRS, Universit´e de Versailles, France, 33, 2001.

T. G. Crainic, M. Toulouse. *Parallel Metaheuristics*. Centre de recherche sur les transports Université de Montréal, Canada, 53, 1997.

K. Dasgupta, M. Kukreja, K. Kalpakis. T*opology-aware Placement and Role Assignment for Energy-efficient Information Gathering in Sensor Networks*. ISCC'03, 2003.

E. Falck, P. Floréen, P. Kaski, J. Kohonen, P. Orponen. *Balanced Data Gathering in Energy-Constrained Sensor Networks*. Algorithmic Aspects of Wireless Sensor Networks. p. 50-70. Springer-Verlag, LNCS3121, Berlin, 2004.

S. Flint. *Algorithm Development & Comparison for Variable Camshaft Timing Optimisation*. Dissertation work, University of Central England, 2004.

P. Floréen, P. Kaski, J. Kohonen, P. Orponen. *Exact and Approximate Balanced Data Gathering in Energy-Constrained Sensor Networks*. Theoretical Computer Science, Elsevier Science, 2005.

J. Florián. *Rozložení vztlaku po rozpětí křídla* I. VAAZ, 1963.

T. Fogarty, R. Huang. *Implementing the genetic Algorithm on Transputer Based Parallel Processing Systems*. Parallel Problem Solving from Nature, p. 145–149, 1991.

Ch. Gagné, M. Parizeau, M. Dubreuil. *The Master-Slave Architecture for Evolutionary Computations Revisited*. Proceedings of the Genetic and Evolutionary Computation Conference, Chicago, IL, 2:1578-1579, 2003.

A. Geist, A.Beguelin, J. Dongarra, W. Jiang, R. Manchek, V Sunderam. *Pvm: Parallel virtual machine*. A user's guide and tutorial for netowrked parallel computing. MIT Press, Cambridge 1994.

D. E. Goldberg, K. Deb, J. H. Clark. *Genetic algorithms, noise, and the sizing of populations*. Complex Systems, vol. 6, p. 333–362, 1992.

D. E. Goldberg. *Genetic and evolutionary algorithms come of age*. Communications of the ACM, vol. 37, number 3, p. 113–119, 1994.

J. J. Grefenstette. *Parallel adaptive algorithms for function optimization*. Report No. CS-81-19, Vanderbilt University, TN, 1981.

P. Grosso. *Computer Simulations of Genetic Adaptation: Parallel Subcomponent Interaction in a Multilocus Model*. PhD thesis, Department of Computer and Communications Sciences, University of Michigan, 1985.

G. Harik, E. Cantú-Paz, D.E. Goldberg, B.E. Miller. *The gambler's ruin problem, genetic algorithms, and the sizing of populations*. In Proceedings of 1997 IEEE International Conference on Evolutionary Computation, p. 7–12, IEEE Press (Piscataway, NJ), 1997.

R. Hauser, R. Männer. *Implementation of standard genetic algorithm on MIMD machines*. In DAVIDOR Y., SCHWEFEL H.-P., MÄNNER R., Eds., Parallel Problem Solving fron Nature, PPSN III, p. 504–513, Springer-Verlag (Berlin), 1994.

E. L. Houghton, P. W. Carpenter. *Aerodynamics for Engineering Students*. Fifth edition, Elsevier Butterworth-Heinemann, Oxford, 2003. ISBN 0-7506-5111-3.

B. Krishnamachari, F. Ordóñez. *Analysis of Energy-Efficient Fair Routing in Wireless Sensor Networks Through non-linear Optimization*. Proc. IEEE 58th Vehicular Technology Conference, Vol. 5, 2844-2848, IEEE Computer Society, Los Alamitos CA, 2003.

J. Lampinen. Differential Evolution – *New Naturally Parallel Approach for Engineering Design Optimization*. In H. Barry, V. Topping (eds.), Developments in Computational Mechanics with High Performance Computing, Civil-Comp Press, Edinburgh, 1999.

S. C. Lin, W. Punch, E. Goodman. *Coarse-Grain Parallel Genetic Algorithms: Categorization and New Approach*. Sixth IEEE Symposium on Parallel and Distributed Processing, IEEE Computer Society Press (Los Alamitos, CA), October 1994.

B. Manderick, P. Spiessens. *Fine-Grained Parrallel Genetic Algorithms*. Third International Conference on Genetic Algorithms, San Mateo, CA, 428-433, 1989.

F. J. Marín, O. Trelles-Salazar, F. S. Hernández. *Genetic algorithms on LAN-message passing architectures using PVM : Application to the routing problem*. Parallel Problem Solving from Nature, PPSN III, p. 534–543, Springer-Verlag (Berlin), ISBN:3-540-58484-6. 1994.

M. Munetomo, Y. Takai, Y. Sato. *An efficient migration scheme for subpopulation-based asynchronously parallel genetic algorithms*. Proceedings of the Fifth International Conference on Genetic Algorithms, page 649, Morgan Kaufmann (San Mateo, CA), 1993.

H. Mühlenbein, M. Schomisch, J. Born. *The Parallel Genetic Algorithm as Function Optimizer.* Proceedings of the Fourth International Conference on Genetic Algorithms, Morgan Kaufmann (San Mateo, CA), 1991.

L. Nolle, I. Zelinka, A.A. Hopgood and A. Goodyear. *Comparison of an self-organizing migration algorithm with simulated annealing and differential evolution for automated waveform tuning*. Advances in Engineering Software, Volume 36, Issue 10, October 2005, pp 645-653, ISSN: 0965-9978.

Z. Oplatková, I. Zelinka. *Investigation on Artificial Ant using Analytic Programming.* GECCO 2006, Seatle, WA, USA, 8-12.7.2006, 11th conference.

A. Oyama, S. Obayashi, T. Nakamura. *Real-Coded Adaptive Range Genetic Algorithm Applied to Transonic Wing Optimization*. Parallel Problem Solving from Nature VI., Paris, France, 712-721, 2000.

C. C. Pettey, M. Leuze, J. J. Grefenstette. *Genetic algorithms on a hypercube multiprocessor.* Hypercube Multiprocessors 1987, p. 333–341, 1987(a).

C. C. Pettey, M. Leuze, J. J. Grefenstette. *A Parallel Genetic Algorithm*. Proceedings of the 2nd International Conference on Genetic Algorithms and Their Applications, San Mateo, CA, 155-161, 1987(b).

W. Rivera. *Scalable Parallel Genetic Algorithms*. Artificial Intelligence Review, 16(2):153-168, 2001.

J. Sarma, K. D. Jong. *An analysis of the effects of neighborhood size and shape on local selection algorithms*. Parallel Problem Solving from Nature IV, p. 236–244, Springer-Verlag (Berlin), 1996.

M. Sefrioui, J. Périaux. *A Hierarchical Genetic Algorithm Using Multiple Models for Optimization*. Parallel Problem Solving from Nature VI., Paris, France, 879-888, 2000.

T. Starkweather, D. Whitley, K. Mathias. *Optimization Using Distributed Genetic Algorithms*. Parallel Problem Solving from Nature, p. 176–185, Springer-Verlag, Berlin, Germany 1991.

R. Storn, K. Price. *Differential Evolution – A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces*. Technical Report TR-95-012, ICSI, 1995.

R. Tanesse. *Parallel genetic algorithm for a hypercube*. Proceedings of the Second International Conference on Genetic Algorithms, p.177–183, Lawrence Erlbaum Associates (Hillsdale, NJ), 1987.

D. K. Tasoulis, N.G. Pavlidis, V.P. Plagianakos, M.N. Vrahatis. *Parallel Differential Evolution*, Proceedings of the IEEE 2004 Congress on Evolutionary Computation (CEC), 2004.

N. Xiao, M. Armstrong. *A Specialized Island Model and Its Application in Multiobjective Optimization*. Proceedings of the Genetic and Evolutionary Computation Conference, 2:1530-1540, Chicago, IL, 2003.

D. Zaharie, D. Petcu. *Parallel implementation of multi-population differential evolution*. CIPC 2003, Concurrent Information Processing and Computing, Nato Advanced Research Workshop, Sinaia, July 2003.

I. Zelinka. *Artificial Intelligence in the Problems of Global Optimization* (Czech Edition), BEN – technická literatura, Praha, 2002, ISBN 80-7300-069-5.

I. Zelinka. *SOMA – Self Organizing Migrating Algorithm.* Chapter 7, 33 p. in: B.V. Babu, G. Onwubolu (eds.), *New Optimization Techniques in Engineering*, Springer-Verlag, 2004, ISBN 3-540-20167X.

I. Zelinka, Z. Oplatková, L. Nolle. *Analytic Programming – Symbolic Regression by Means of Arbitrary Evolutionary Algorithms*. In: Special Issue on Inteligent Systems, International Journal of Simulation, Systems, Science and Technology, Volume 6, Issue 9, August 2005, pages 44 – 56, ISSN 1473-8031.

I. Zelinka, E. Volná. *Neural Network Synthesis by Means Of Analytic Programming - Preliminary Results.* In: Proc. 11th International Conference on Soft Computing Mendel'05, Brno, Czech Republic, 2005, ISBN 80-214-2135-5.

I. Zelinka. *Investigation on Realtime Deterministic Chaos Control by Means of Evolutionary Algorithms.* In: 1st IFAC Conference onAnalysis and Control of Chaotic Systems, Reims, France, June 28-30, 2006.

LIST OF FIGURES

Figure 3-1: An example of master-slave parallel GA. The master stores the population, executes the selection, crossover and mutation process and distributes single individuals to slaves. The slaves only evaluate the fitness of the individuals....12
Figure 3-2: An example of fine-grained parallel GA. This class of parallel GAs has one spatially-structured population and can be efficiently implemented on massively parallel computers...13
Figure 3-3: An example of multiple-population parallel GA. Every process is a standalone GA which sometimes exchanges an individual with its neighbours. 13
Figure 3-4: An example of hierarchical parallel GA combining the island model on higher level and fine-grained GA on the lower level....22
Figure 3-5: Scheme of an hierarchical parallel GA. While the top level is considered as the island model, the lower level is represented by a master-slave GA....22
Figure 3-6: An example of hybrid parallel GA with the island model on both levels. At the lower level, the migration rate is higher and the communication topology denser than at the upper level....23
Figure 4-1: Cluster structure including interconnection of its inner modules....27
Figure 4-2: Schema of server structure....30
Figure 4-3: Schema of terminal structure....35
Figure 4-4: Data classes that are transported within ClusterData....39
Figure 4-5: Main page of cluster's administration interface. Here you can see state of a parallel project, start and stop time, CPU time used for processing this assignment and also the speed-up ratio. In addition to this, it is possible to modify the project properties, to start/terminate or create new parallel project....41
Figure 4-6: Creating new project....42
Figure 4-7: Project settings. Here you can add and remove classes/files to/from a project....42
Figure 4-8: Terminals list and their statistics....43
Figure 4-9: TimeCluster – test results....53
Figure 5-1: Diagram of integration UDPOptCluster into SOMA implemented in Matlab or Mathematica and its communication with other terminals. 62
Figure 5-2: Spreading of information in virtual network of terminals. Influence of a 'good' leader extends in steps across the entire grid....63
Figure 5-3: Performance comparison of various parallelisation models....66
Figure 6-1: TIVCT engine installation in Visteon....68
Figure 6-2: Block encapsulating model of the optimised engine and representing an interface

between SOMA and the model......69
Figure 6-3: Block schema of the optimised engine......70
Figure 6-4: Examples of relay node placement experiments to a regular grid of sensors. Gray circles represent sensor nodes, small red squares are relay nodes and the large grey node at the south side of the area is a sink where all data is collected......76
Figure 6-5: Balanced data rate F0.5 as a function of the number of relay nodes. Results for array with 100 source nodes placed in regular grid 10x10 without any obstacles in the landscape . 77
Figure 6-6: Examples of relay node placement experiments on a network with 36 randomly placed sensor nodes. Gray circles represent sensor nodes, small red squares are relay nodes and the large grey node at the south side of the area is a sink where all data is collected......79
Figure 6-7: Balanced data rate F0.5 as a function of the number of relay nodes. Results for array with 36 randomly placed source nodes with obstacles in the landscape......80
Figure 6-8: The horseshoe vortex......81
Figure 6-9: The simplified horseshoe vortex......81
Figure 6-10: The relation between spanwise load variation and trailing vortex strength......82
Figure 6-11: Elliptic loading......83
Figure 6-12: Typical spanwise distribution of lift......85
Figure 6-13: Loading make-up by selected sine series......87
Figure 6-14: Lift-versus-incidence characteristic for an aerofoil section of a certain profile......89
Figure 6-15: Modified parameters of a wing......93
Figure 6-16: Aeroplanes from Evektor production: a two-seated very light aeroplane “Harmony” (left) and the four-seated VUT100 Cobra (right). Source: www.evektor.cz......95
Figure 6-17: SportStar: shape of evolved wing and its lift characteristics......96
Figure 6-18: VUT-100 Cobra, approach I: shape of evolved wing and its lift characteristics......98
Figure 6-19: VUT-100 Cobra, approach II: shape of evolved wing and its lift characteristics......99
Figure 6-20: VUT-100 Cobra, approach III: shape of evolved wing and its lift characteristics......100
Figure 6-21: VUT-100 Cobra, the dead end: shape of evolved wing and its lift characteristics......101

LIST OF TABLES

Table 4-1: TimeCluster – test results......52
Table 5-1: Training set for neural network and its real output used as a time-demanding objective function example for the process of somalisation......65
Table 5-2: Overview of parallel SOMA test results......66
Table 6-1: Output of the engine optimisation process......71
Table 6-2: SportStar: wing properties......95
Table 6-3: SportStar: overview of optimised wing parameters......96
Table 6-4: VUT-100 Cobra, approach I: overview of optimised wing parameters......97
Table 6-5: VUT-100 Cobra, approach I: wing properties......98
Table 6-6: VUT-100 Cobra, approach II: overview of optimised wing parameters......98
Table 6-7: VUT-100 Cobra, approach II: wing properties......99
Table 6-8: VUT-100 Cobra, approach III: overview of optimised wing parameters......100
Table 6-9: VUT-100 Cobra, approach III: wing properties......100
Table 6-10: VUT-100 Cobra, the dead end: overview of optimised wing parameters......101
Table 6-11: VUT-100 Cobra, the dead end, approach III: wing properties......101

LIST OF PUBLICATIONS

Miroslav Červenka. Optimization of a GM Combustion Engine. Internal report, classified. Glasgow, United Kingdom, 2004.

Miroslav Červenka. Distribuované evoluční algoritmy. Proceedings Počítačové architektúry a diagnostika PAD'04, Moravany nad Váhom, Slovak Republic.

Miroslav Červenka, Pavel Stříž, Martin Štěpánek. Instalace WWW serveru na lokálním počítači. Proceeding of the 1st Bata's International Conference for Graduate Students. Zlín, UTB, 2005. ISBN 80-7318-257-2.

Miroslav Červenka, Pavel Stříž, Martin Štěpánek. Můj počítač, můj server. Proceedings of the 7th conference „Internet a konkurenceschopnost podniku". Zlín, UTB, 2005. ISBN 80-7318-269-6.

Miroslav Červenka, Ivan Zelinka. Parallel Computation Platform for SOMA. Proceedings of the 19th European Conference on Modelling and Simulation „Simulation in Wider Europe", ECSM 2005, Riga, Latvia 2005, ISBN 1-84233-112-4

Miroslav Červenka, Ivan Zelinka. Parallel Computation Framework for SOMA. Proceedings of the 9th International Research/Expert Conference „Trends in the Development of Machinery and Associated Technology", TMT 2005, Antalya, Turkey, 2005, ISBN 9958-617-28-5

Miroslav Červenka, Ivan Zelinka. Relay Node Placement in Energy-Constrained Networks Using SOMA Evolutionary Algorithm. 32nd Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2006), Měřín, Czech Republic, 2006

Miroslav Červenka, Ivan Zelinka. Relay Node Placement in Energy-Constrained Networks Using SOMA Evolutionary Algorithm. The IASTED International Conference on Artificial Intelligence and Applications (AIA2006), Innsbruck, Austria, 2006, ISBN 0-88986-558-2

Miroslav Červenka, Ivan Zelinka. Relay Node Placement in Energy-Constrained Networks Using SOMA Evolutionary Algorithm. International Conference in Applied Mathematics for Undergraduate and Graduate Students (ISCAM 2006), Bratislava, Slovak Republic, 2006

CURRICULUM VITAE

Name:	Miroslav Červenka
Present address:	Budovatelská 4805
	Zlín 760 05
	Czech Republic
E-mail:	miroslav.cervenka@centrum.cz
Date of birth:	20. 04. 1980
Marital status:	single

Education:

09/1998 – 06/2003	Tomas Bata University in Zlín, Faculty of Technology, master degree in Automation and Process Control in Consumer Industry.
09/1994 – 05/1998	Technical Secondary School, branch "Low Voltage Electrical Engineering", specialization "Electronic Computer Systems".
Languages:	Czech: mother tongue, English: active listening and writing, German: active listening, speaking fair.

Professional experience:

2003 – 2005	teaching courses at TBU (Applied Informatics, Telecommunications)
04 – 08/2004	Study at University of Strathclyde, Glasgow, UK. Application of parallel evolutionary algorithms (combustion engine opt.)
06 – 08/2005	Professional traineeship at Helsinki University of Technology, Helsinki, Finland. Optimization of message routing and relay node placement in wireless sensor networks.
01 – 05/2006	Research-fellowship in Evektor spol. s r. o. Evolutionary design, aerodynamic and structural optimization of a wing geometry.

Research and other activities during studies:

2002 – 2004	Cooperation on the evolution of gas micro-flow sensor. Included C++, Java, ASM, driven on i51 and XEMICS microcontrollers.
12/2002	Award for 2nd place in "SVOČ2002" student science competition.

world domination proceeds as planned