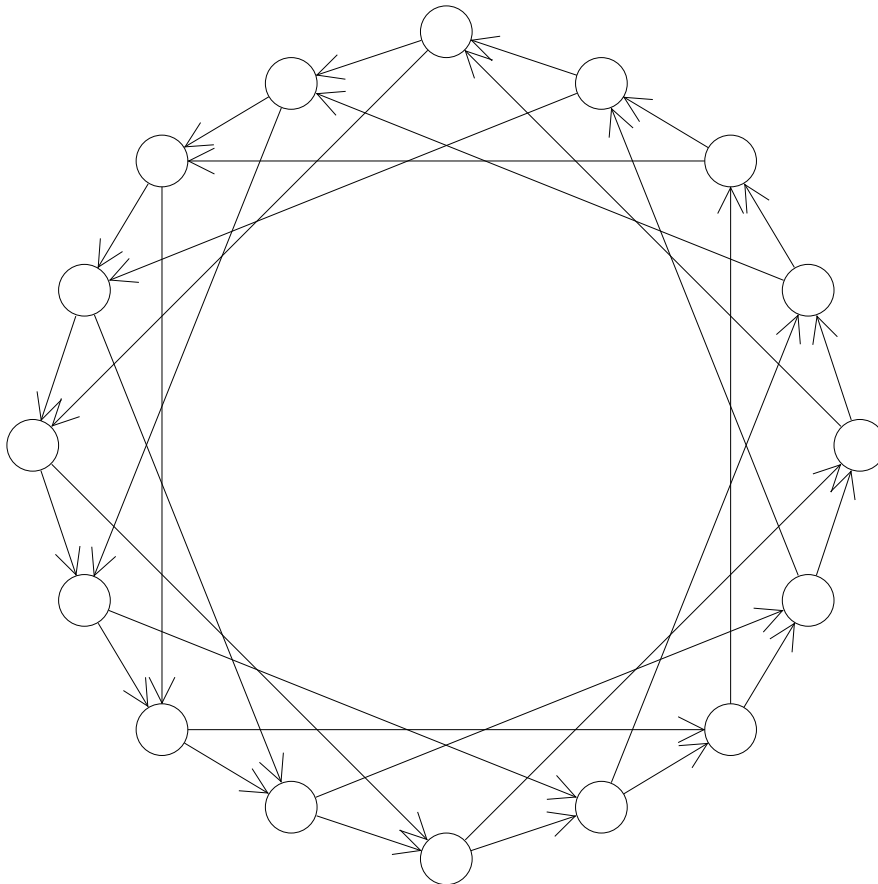


# Scheduling parallel processes using Genetic Algorithms



by Martijn Meijer  
Supervised by Dick van Albada  
Master thesis in the field of Artificial Intelligence,  
section Autonomous Systems  
24th February 2004





# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
<b>2</b>	<b>Parallel computing</b>	<b>7</b>
2.1	Goals . . . . .	7
2.2	Continuous time simulation . . . . .	7
2.2.1	An example: Weather forecast . . . . .	8
2.2.2	Global solutions to performance shortage . . . . .	9
2.3	Parallel computing theory . . . . .	9
2.3.1	Scalability . . . . .	10
2.3.2	Behaviour patterns . . . . .	10
2.3.3	Amdahl's law . . . . .	12
2.4	Performance optimisation . . . . .	12
2.4.1	Performance constraints . . . . .	13
2.4.2	Competition for resources . . . . .	13
2.4.3	Load balancing . . . . .	15
<b>3</b>	<b>Scheduling</b>	<b>17</b>
3.1	Layer models . . . . .	17
3.1.1	Platform vs. application . . . . .	18
3.1.2	Portability Layer . . . . .	18
3.1.3	Run-time environment . . . . .	19
3.2	Monitoring . . . . .	19
3.3	Difficulties with migration . . . . .	20
3.3.1	Checkpointing and restarting . . . . .	20
3.3.2	Communication . . . . .	22
3.3.3	Open files . . . . .	22
3.4	Rescheduling . . . . .	22
3.5	Search techniques . . . . .	23
3.5.1	Breadth/depth first . . . . .	23
3.5.2	Best first/hill climbing . . . . .	24
3.5.3	Simulated annealing . . . . .	25
3.5.4	Genetic algorithms . . . . .	25

<b>4</b>	<b>Genetic algorithms</b>	<b>27</b>
4.1	Overview . . . . .	27
4.2	Chromosome representation . . . . .	27
4.3	Fitness and selection . . . . .	28
4.4	Diversification . . . . .	30
4.5	Design choices . . . . .	31
<b>5</b>	<b>Performance model</b>	<b>33</b>
5.1	Cluster model . . . . .	33
5.2	Work measure . . . . .	34
5.3	Optimisation methods . . . . .	35
5.3.1	optPerProgramme . . . . .	35
5.3.2	lessStupidOpt . . . . .	36
5.3.3	doubleOpt . . . . .	36
5.4	Modelled clusters . . . . .	36
5.4.1	test-input7b . . . . .	36
5.4.2	test-input9 . . . . .	38
5.4.3	test-input16d . . . . .	39
<b>6</b>	<b>Results</b>	<b>41</b>
6.1	Different approaches . . . . .	41
6.2	Preliminary setting of parameters . . . . .	42
6.3	Selection methods . . . . .	45
6.4	Different models . . . . .	46
6.5	Number of generations/population size balance revisited . . .	49
6.6	Alternate differentiation methods . . . . .	51
6.6.1	Swap mutation . . . . .	53
6.6.2	Scaled mutation . . . . .	53
6.6.3	Uniform crossover . . . . .	55
6.7	Dependency between population size and mutation probability	55
6.8	Comparing the final genetic algorithm with other scheduling methods . . . . .	57
<b>7</b>	<b>Conclusion</b>	<b>61</b>
7.1	Research findings . . . . .	61
7.2	Remaining work . . . . .	62
7.2.1	Validation of the model . . . . .	62
7.2.2	Rescheduling . . . . .	62
7.2.3	Simulated annealing . . . . .	62
7.3	Final words . . . . .	62

# Chapter 1

## Introduction

Parallel computing is a prerequisite for solving many large computing problems in a limited time frame when the performance of a single machine is not sufficient for the problem. For some part, this computing capacity will be provided as dedicated clusters and/or parallel machines. However, as most regular workstations at a research facility are often not being used to the fullest of their capacity, additional capacity can be obtained. The performance of such a non-dedicated, mostly heterogeneous cluster depends on the computational power of the individual nodes, the communication speeds between them, and the background load due to normal use.

Optimal scheduling of parallel programs in such an environment is a very complex problem. If a computer in such a non-dedicated cluster is needed for its normal use, it may be necessary to release the workstation from some of its tasks in the cluster as needed. Also, workstations might be added (back) to the cluster, and new parallel programs might be added to be run on the cluster. To do this, tasks need to be able to move from one node to another. Finding the optimal solution takes much more time compared to finding a solution that is not optimal, but good enough.

Using standard search techniques [13], a shallow search should be sufficient, but in some cases this will result in a greatly non-optimal situation. Since these straightforward search techniques for finding a solution to these problems could easily get stuck at local optima, artificial intelligence search techniques can prove to be very useful for this problem because of their ability to overcome such local optima.

Genetic algorithms [12] have been around for about 35 years now. Based on the idea of natural selection by Darwin, they involve a population which changes each time step (generation), where the fittest have a higher chance of producing offspring than others.

This master thesis will try to answer the question if genetic algorithms are suited to create better parallel computing schedules than current methods are able to. This is done in relation to the Dynamite project [7, 10,

8, 9].

There are some different strategies in genetic algorithms [11, 14, 4], and experiments will be done in order to find the best suited one for this problem.

In chapter 2, I will talk about some of the concepts of parallel computing, why it is used and what the problems are.

Next, in chapter 3, I will focus on the problem at hand, and show how current implementations are able to move tasks around without changing the tasks themselves.

Chapter 4 covers genetic algorithms, and the different methods are discussed.

In chapter 5, the model I have used in my experiments is introduced, and the modelled clusters are described.

The results of the experiments are covered in chapter 6.

Finally, the conclusions of the research are presented in chapter 7, as well as some work that remains to be done.

## Chapter 2

# Parallel computing

In this chapter I will explain what parallel computing is, why it is necessary in some fields of research, what is needed for parallel computing and what the general problems are.

### 2.1 Goals

Since the beginning of computer history, there have always been problems that required more computing power than available from a single computer, and there was either no time or no patience to wait until faster computers were developed and acquired. This applies to many fields of research, ranging from weather prediction to cancer research and atomic fusion simulations.

### 2.2 Continuous time simulation

Continuous time simulation is an often-used method to simulate various situations, including chemical, biological, physical or logistical situations. In order to predict various real-life situations, simulation is sometimes needed to predict the result of a potentially costly or dangerous change in the situation. These simulations sometimes (depending on the complexity of the situation) require a great deal of computing power.

As a truly continuous time simulation is impossible because of the non-discreteness of time and space, time is discretised into time slices. These time slices should be small enough so that in every time step each of entities in the simulation (in our weather prediction example in 2.2.1, water or air in a single sector) can be said to be independent of the other parts, and can be calculated on its own.

Implementing such a single entity is usually easy when someone with adequate knowledge of the real-life situation is willing to cooperate. To such an expert in the field it may be trivial in what way the situation changes for a single entity, while the problem in its entirety may be too large or too

complex to grasp. This way the situation has been broken up into little pieces.

After all parts are done calculating for the time step, the communication between the entities (representing the influence between the parts) is done and time is advanced one step.

In principle this is a serial approach. However, since the various entities can be implemented as different tasks running on different nodes, it is easily parallelisable. The resulting behaviour is usually lock-step (we'll come to that in section 2.3.2).

The simulation itself needs to be validated against the real-life situation in order to be sure that the simulation will give a correct prediction of situation that haven't occurred in real-life (yet). This validation is done with situations which have already occurred, in order to be able to check the results. This is to see if:

- the single entity is implemented correctly,
- the entities work together in a correct way, and
- the time, geometrical and (if applicable) other slices are chosen small enough.

### 2.2.1 An example: Weather forecast

Let us introduce an example we will be referring to later and which is a conversation starter around the globe: what will the weather be like tomorrow? Weather forecasting is very time-critical: there is no sense in calculating the weather for tomorrow and having the result ready early next week.

Weather forecasting could be achieved as follows: various properties of the current weather around the world are known, like temperature, air pressure, humidity, wind direction and speed, and at sea the water height, direction and speed of the flow and temperature of the water. Combine this information with information like the gravitational pull on the seas from the sun and moon and the physical properties of water and air, and you have a large dynamic model.

Lets say we divide the globe into sectors with a certain width, breadth and height. Each sector has a certain influence on the parts surrounding it. Non-adjacent sectors should not influence each other directly, which is achievable if the time-step chosen is small enough, so that changes (for instance, pressure) cannot pass the sector in a single time-step (with the medium of air, the speed of sound would be sufficient).

The precision of the forecast will, apart from the quality of the simulation model, depend on the number (and size) of sectors, the size of the time-step used and the quality of the measurements.

Since the scalability of this problem is 4D (width, length, height (!) and time), making forecast with double accuracy might involve 16 times as many calculations.

### 2.2.2 Global solutions to performance shortage

There are three solutions for the lack of available sufficient computing power: first, conceive better computing strategies and algorithms. Usually this is done extensively, but in the end there are always limits in how far things can be optimised.

Second, one can convince the management that faster, more expensive, computers are needed. Sometimes this is no option, since there simply are no faster computers on the market than those being used. Waiting until faster computers are developed and available is usually no option.

Last, and what will be focused upon in this thesis: optimise the way the current computing power is distributed. Generally, management will not approve the replacement of a workstation which is operational and has not been written off yet. However, when a new state-of-the-art computer is acquired for someone with low computing requirements (for instance a receptionist), it will usually have more computing power than is needed. Since this computer is being idle most of the time, it would benefit research if that power would be available to speed up the calculations in order to make the deadline.

## 2.3 Parallel computing theory

Parallel computing is the simultaneous and coordinated use of multiple computers to solve a single problem, where simultaneous use of multiple computers is understood to be different computers working at the same time, each on a different part of the total problem.

Parallel computing is generally used to solve problems that are time critical, when, it would take too much time on a normal sequential machine to reach a conclusion, sometimes before a strict deadline. As we've already mentioned in the weather forecast example, calculating the weather for tomorrow implicates a deadline sooner than that.

Another example would be a very large simulation which would take months and could effectively put the rest of the research on hold for quite some time. This is not a clear deadline, but shows that even without a deadline, fast calculation might still be needed.

The memory requirements to solve problems which use more memory than a single computer can manage are another reason for using parallel computing. In weather prediction, all information of the weather in each part of the map together might be too much for a single computer to have in memory. Dividing the data across multiple computers solves this problem.

In order to be successful, a problem should be easily divisible so that every computer (in parallel computing often called 'node') is able to calculate a 'piece of the puzzle'. The specific instructions for a single 'piece' are called a task. In the example of the weather forecast this might be calculating the weather for a smaller part of the globe. This task should be able to run without being very dependent on the other tasks. However, you can't expect the tasks to be completely independent, so they should have the possibility to communicate. In the example, this could be information like temperatures, air pressure and such, exchanged between neighbouring sectors.

### 2.3.1 Scalability

To what extent it is possible for a program to run on any number of nodes is called scalability. If a problem is ultimately scalable, it will run  $n$  times faster on  $n$  nodes, and a  $n$  times larger version of the program will run equally fast on  $n$  times as many nodes.

The scale of divisibility of a parallel process is very important for scalability. Depending on the problem, the sizes of the tasks relative to the size of the total problem can be anywhere from very small, which might result in a very scalable program, up to just a single monolithic serial program.

You can have a problem which is very divisible in hundreds or thousands of separate small tasks, which would be very convenient for scheduling. This is because if there is a small amount of CPU-time left on a node, a small task can be added, as long as the computational gain significantly exceeds the scheduling and (if needed and possible) migration costs. This is not possible when you have a problem with just two large pieces.

For instance, it is not practical to have a node calculate the weather in a single province, since that limits the number of nodes to be used to the number of provinces. A better approach is to have squares of a certain geographical size. That way it is possible to adjust the size if the program is run on a larger or smaller cluster.

Granularity is the level of independence of the calculation on the communication. If a process can only make a very small calculation before it has to wait for communication with other processes, it is called fine grained. Course grained processes, which act much more independently of other tasks, are less likely to stall waiting for communication. Therefore fine grained processes will yield a lower speedup, as will be discussed in section 2.3.3.

### 2.3.2 Behaviour patterns

Depending on the problem which is to be solved, the behaviour of the parallel program can have a distinct pattern. I will discuss some of the behaviours here [16].

One example is embarrassingly parallel where all nodes receive an amount of work right at the start (after some performance measurements) and - if the performance of the nodes doesn't change during computation - finish at approximately the same time. There is no communication at all between the tasks. This is of course only possible in a situation where the calculations are independent of each other, but will give a performance boost close to linear; when using 3 (equal, dedicated) nodes instead of 1, it will be finished in approximately 1/3 of the time.

Another situation is the farming approach. Here the nodes request a task from a master node, which they can calculate independently of the other tasks. The difference with embarrassingly parallel is that in this case, the tasks are handed out on a need-to-have basis, instead of being assigned at the very beginning. For example, let's say you have to do a lot of very different calculations, on data which is not influenced by other tasks. Every node requests a calculation, computes the answer and returns the answer to the master node. One of the benefits of this approach is that it is very dynamic in its nature. When a node fails, the same task can simply be taken care of by another node, and when a node is added, it just requests a task, like all other nodes do when they are finished with their previous task. Performance boost will be near-linear but again, this approach is only possible with a few problems. Popular parallel projects like SETI@Home and Distributed.net work in this way.

At the other extreme is lock-step, in which all nodes do a small amount of work at the same time, after which they all communicate and start synchronously on the next amount of work. This is called lock-step because the tasks have a synchronised step clock, and will have to wait for the task which takes longest. The performance boost will depend on the difference in the time the tasks take, plus the communication overhead. An example for lock-step would be the weather forecast. A task can only proceed calculating the next time-step if the previous time-step has been finished on all neighbouring tasks and all results are sent back and forth.

Furthermore, there is pipelining. Here every node does a different step in the calculation process. For instance, when calculating  $2 \times 3 + 4$ , the first node would calculate  $2 \times 3$  and after that, the second node would calculate  $6 + 4$ . If you have a stream of the same calculations, of which only the data is different, the first node is always one calculation ahead of the second. In such pipelines computation and communication can easily be overlapped. So this would be (in an situation with a very large amount of data) nearly linear faster than when using a single node. However, if the intermediate results (per single calculation) are very large, the communication will limit performance because a lot of time will be spent on sending large blocks of data instead of the calculation itself.

Sometimes a serial program has already been written, and has had a quick and dirty adaption to use a parallel system. In this case, the master

program is just a serial program which now and then outsources a calculation of which it won't need the results for some time. How much can be outsourced, depends on the problem and the way the serial program was written. The performance gain depends on how much can be outsourced, but is generally relatively low.

This summary is nowhere near complete, but it gives a general idea of how different the behaviour might be.

### 2.3.3 Amdahl's law

Parallel computing is, in some behaviours, dependent on a central housekeeping process. This housekeeping process is typically not paralisable. As Amdahl wrote some time ago [1], the achievable speedup is as follows:

$$Speedup = \frac{1}{s + \frac{p}{n}}$$

where  $s + p = 1$ ,  $s$  and  $p$  representing the portion of time taken by respectively the sequential and the parallel parts in the program and  $n$  is the number of nodes used.

Therefore, it is important to minimise the sequential portion of the program, so more nodes can be added without having to wait for the housekeeping process, otherwise computing cycles will simply go to waste.

In embarrassingly parallel this is not the case since there is almost no serial part. The problem is divided at the very start, and from that moment on, it will work completely independent of the other processes.

Also, as Gustafson wrote [6], Amdahl's law may not apply in some cases of simulation, where the size/resolution of the problem may be scaled with the number of processors, keeping the run time mostly constant. This has no real significance here, since it is not the object of this thesis to scale the problems when nodes become less or more available.

## 2.4 Performance optimisation

One of the central problems in many aspects in parallel computing, is performance optimisation. Performance optimisation can be defined as minimising some cost function constrained by predefined boundaries.

Performance is measured by determining how well the performance goal is obtained. The performance goal is one specific parameter that has been chosen to be either as small or as large as possible, while changing the other parameters in order to achieve that goal.

For instance, the performance goal can be to minimise the total turn-around time (the time it takes before all tasks are finished), to minimise the turn-around time of one specific program, or (as discussed above) the precision of the calculation of a program.

### 2.4.1 Performance constraints

Performance of a task is held back by constraints, typically limited CPU time, limited memory size and speed, limited communication bandwidth, latencies in communication and various limitations with input/output devices.

Generally, the performance of a task is limited by a single constraint specifically. This constraint is then called the bottleneck. For example, if a task needs to send an lot of data over the network, and has an average amount of calculations to do, and it is placed on a fast node with a slow network connection, the network will be the bottleneck. If that same task is moved to a node with a very fast network connection but a very slow processor, the network may not be the bottleneck any more. The processor might now be too slow to keep up, relative to the needs of the task. The processor has then become the bottleneck.

Because the bottleneck is by definition the main factor holding back performance, it makes little sense to move the task to a node that isn't better at the bottleneck constraint, even if that node is better at the non-bottleneck constraints. If a task is held back by the network, having a faster processor will not help. It was held back by the bottleneck, and remains being held back by the bottleneck.

So, the wise thing to do is to run the task on a computer where the performance of the bottleneck-constraint is better; in the previous example a computer with more network bandwidth. As this bottleneck property is enhanced, the bottleneck is loosened and the performance of the task will improve. If the bottleneck constraint is enhanced even further, the bottleneck is removed and the performance will be held back by another constraint instead (in the example: the CPU speed). As one can see, there will always remain a bottleneck.

In a more or less optimal situation, it will not be very clear what the actual bottleneck is, or the bottlenecks will keep changing during the execution of the task. This way, all resources will be used to the fullest of their capacity, and few will go to waste. Also when all tasks have the same bottleneck that can simply not be enhanced any further, no better result can be obtained.

On a different level, parallel programs (see section 2.3.2) are being held back by bottleneck tasks in much the same way. Either a single task is the slowest one, holding back the other tasks, or in a more balanced situation, a few are more or less equally slow.

### 2.4.2 Competition for resources

Since the four important resources in a computer (CPU-time, memory, input/output and communication) are all limited, there is a competition for

these resources between the various tasks running on that node.

When CPU-time is amply available, it is given to a task simply when it is required. If there is not enough available, all tasks may get equal portions, or they may get unequal scaled portions if some tasks have higher priorities. If a task doesn't require all of the portion it has been granted, the remainder of the portion is divided among the tasks that require more than their portion provides for.

Memory is a simple matter: if a task requests memory, and it is available, it is granted, and otherwise it is denied. It is up to the programmer to make sure the program can handle such situations. Using virtual memory (using a part of the hard disk as an extension of the memory) is possible, but because of the extreme difference in performance (seek time is typically 200.000 times higher than when using the main memory), this is usually not an option.

Input/output (I/O) is no simple matter because it can have very different characteristics. I/O usually has very high latencies (compared to latencies of the memory and within a processor): 10 milliseconds random access time is not unusual for a hard drive, versus 50 nanoseconds for memory. Usually the process which requested I/O will be removed from the processor by the time the hard drive has even begun to transfer the first byte of data. For this reason hard disks read ahead, reading more than they actually should. The extra data is kept in cache and can be accessed very quickly when requested. If this were not done, performance would be really bad if two tasks tried to read or write to the same drive: between every read, the heads of the hard drive need to be re-aligned.

Communication is the most complex of the four. It has 2 main parameters: network delay, which is the time between when a packet of data is starting being sent and when it is starting being received, and bandwidth, which is a measure for how much data can be transferred in a certain amount of time.

The delay is influenced by many factors. Roughly, it is the sum of all the delays of the lines in a route from source to destination. Each line has a delay depending on the physical length and its interface to the other lines. It is possible that there is more than one route from source to destination; the slower route may be used if the fast route becomes congested (effectively becoming the slower one): an interface may become saturated if too much information from various nodes needs to be send across, in which case packets can be drastically delayed or even get lost, in which case they need to be resent.

The bandwidth is simply limited by the slowest line, where a fast line with its bandwidth divided for a large number of tasks can be seen as a slow line. In each line, when saturated, the bandwidth is divided among the connections running over that line. Again, if the interface to the other lines is overloaded, packets may be lost and the performance of the line might

diminish<sup>1</sup>.

### 2.4.3 Load balancing

Since the nodes can have very different performance characteristics, and programs can have very unbalanced resource requirements (extreme example: a two-task program where one task is doing nothing except waiting for the other task to complete), something needs to be done in order to make sure that the tasks are distributed among the nodes so that the performance is maximised, for instance by minimising the total turn-around time (the time taken before the answer is returned to the user). This is called load balancing.

With two tasks, one processor-intensive, and another processor-unintensive, and 2 nodes, one faster than the other, the processor-intensive task should (obviously) run on the faster one.

Another example. Two tasks are very communication-intensive, with many short messages, and not very processor-intensive. If these tasks are placed apart from each other, they end up spending more time waiting for communication than calculating. If these were to be placed on a single node, communication delay would be close to zero, and the problem will be finished sooner.

With these kinds of sizes and complexities the optimal solutions are pretty straightforward, with tens or hundreds of tasks it becomes quite non-transparent what solution is best, especially when the communication and other resources and constraints need to be taken into account.

### Dynamic load balancing

When using a non-dedicated cluster (for example, a cluster of workstations on which users can log in and run applications), the performance and the behaviour of the nodes can change dramatically over time. Also, the behaviour of the tasks can change over time. In either case, dynamic load balancing is needed, which will change the schedule accordingly.

For example, there is a parallel lock-step program running. Somebody logs onto one of the nodes, and starts running a big application. Let's assume that the parallel task on that node is processor-intensive. In this situation, the node becomes relatively slower, the processor becomes the bottleneck for the task, the task becomes the bottleneck for the program and tasks running on other nodes will have to wait before they can communicate with the tasks running on that node, resulting in a big performance drop. In such a case, some (maybe all) tasks should be moved from the node to other nodes that might be relatively light-loaded.

---

<sup>1</sup>For more information on competition for network resources, see the section on Congestion control in Tanenbaum's Computer Networks [15].



## Chapter 3

# Scheduling

In this chapter we will further concentrate on the problem at hand: scheduling. We will discuss a possible implementation of parallel systems and some of the principal design choices. We will begin with explaining how a parallel subsystem might be incorporated into an existing system. I will discuss how the general problems are usually solved and what problem I will be looking at.

### 3.1 Layer models

In order to show how a parallel system works, first a few things need to be explained about how a single computer is organised: the inner workings of a computer are layered. A bottom layer has definitions of a physical nature, while a top layer is what the user sees on his/her screen. All layers between are built upon the layer underneath, and are dependent on what the layer below is able to offer. If an underlying layer fails, errors might occur in the layers above, unless special arrangements have been made to overcome such failures.

Perhaps the best-known example is the OSI reference model for networking [15]. The lowest level is the physical layer, it defines the way 0's and 1's are represented as electrical currents.

On top of the first layer, the data link layer makes small frames of hundreds or thousands of bytes and makes sure none of the packets are lost in the process, if necessary resending the frames.

This goes on up to the highest layer, the application layer, which takes care of the highest (to the user most visible) level problems like terminal size, positioning the cursor and such.

Each of these layers is completely dependent on the layer below in order to function correctly. The lower layer might be programmed in a different programming language by a different company. Only the exact way of how to pass information to the other layers, called the interface, needs to be known.

No information on how the lower layer actually does its job is required, although some basic knowledge usually can be helpful to the programmer of the higher layer. Likewise, a layer does not have to know what the information it is sending or receiving represents for the higher layers. It is just a bunch of data it needs to handle in the way it has been instructed to.

The actual workings of the layers is irrelevant, only the interfaces are of interest to the other layers. One can assume that an interface (and with it, all underlying layers) is implemented correctly and should be able to rely on that. How lower layers handle what they do, is essentially none of another layer's business.

### 3.1.1 Platform vs. application

The most obvious (two-)layered distinction in computers is hardware vs. software. Hardware provides certain services (for instance, memory makes sure that information can be stored), which are being exploited by the software in order to do the things the software needs to do.

However, most operating systems provide services in software if the service is not provided in hardware. In the case of memory, an operating system might allocate some of the hard disk as virtual memory. A better distinction would be platform (consisting of hardware and operating system) versus application.

The platform defines the system from an application viewpoint. For the application it doesn't matter if some property of the platform is implemented in hardware or in the operating system. It is up to the operating system to make the system coherent, independent of the underlying hardware. One would like to address a hard disk as just a hard disk, and not necessary as the specific type and brand of hard disk which it is.

### 3.1.2 Portability Layer

Developing software that enables programmers to easily write code that should run unmodified on a variety of hardware-platforms, requires an abstraction layer. This abstraction layer is an extension to the operating system. To the application it is part of the platform, and to the actual platform it is seen as an application. With this layer, applications will interface with the portability layer, oblivious to the underlying actual platform, effectively creating a new platform. Since to the operating system, the portability layer is an application, we'll call this a virtual platform.

When a program calls a function which might not be available on all considered platforms, the call is mapped to a function of the portability layer instead. That function wraps the platform function by either

- converting all relevant data to a format which the actual platform can understand,

- passing the converted data to the corresponding function of the actual platform, and
- converting the result back to the platform-independent format used in the virtual platform,

or, when no corresponding function is available, implement the function in the portability layer itself.

### 3.1.3 Run-time environment

Because the computers available in a non-dedicated parallel cluster are often heterogeneous, a portability layer is a necessity in order to be able to run any task on any node.

The compiled abstraction layer is called run-time environment, or RTE for short. Different platforms will have different versions of the RTE, because the RTE itself is different for each platform. This in contrast to the interface the RTE offers to the applications, which is platform-independent.

This way, a program can address any platform uniformly. However, if the conversion between the formats is complicated, this may have a serious performance penalty. Also, when certain hardware has very specific functions, they will become unavailable: the programmer is programming for a virtual platform now, as provided by the original platform in combination with the RTE.

Of course, hardware and/or operating systems can be built to directly implement the (then no longer virtual) platform, without an RTE. Sun has tried this with the Java Machine, with little success.

Actually, the JVM (Java Virtual Machine) is a good example [3] of a virtual platform. When compiling Java, the code is compiled to so-called Java byte code. This byte code is not very human-readable, and is neither machine-readable. However, the JVM interprets this byte code and converts all functions to platform calls while the program is running. You can also compile the Java byte code to a platform-dependent form, which will make the program, in most cases, significantly faster.

## 3.2 Monitoring

One way of making an estimate of the needs of the running tasks, is to measure how much processor and network resources the tasks are currently using. Retrieving this information is called monitoring. Also, an estimate must exist on how much processor speed and network capacity is available.

One way of retrieving information is by using general system information, for instance the amount of CPU time a task is using. This information is available to any process running on the system, so also to the parallel

subsystem. However, the abstraction layer complicates things here. On some types of computers, certain calls could, relatively to other types of computers, take more time than others. Therefore the CPU times of tasks are not directly comparable to those of tasks running on different platforms.

A different way of retrieving information is through the RTE itself. Since the RTE handles all communication, it seems only logical that it can keep statistics about the tasks' communication. However, in practice this might involve a large overhead with communication-intensive tasks.

### 3.3 Difficulties with migration

One way of changing an unbalanced situation is through migration. Migration is to move one or more tasks to a different node when the parallel system is already running. This should be done in such a way that the parallel program is finished in less time than it would without the tasks being moved, including the time it takes to move the tasks.

In order to be able to make successful and transparent migrations, various technical problems have to be taken care of. From the programs point of view, except for a long delay, the situation will have to seem unchanged before and after the migration.

#### 3.3.1 Checkpointing and restarting

Checkpointing is the process of pausing a task, and the preparation for migration. This includes reading all memory used by the task being migrated, including information about shared libraries being used and the contents of the processor registers, saving the current signal state and such. All this information is saved to a file on disk. This file is called the checkpoint file.

On the target node, the checkpoint file is read, and the original status must be restored. Then the task can be resumed and (if everything is done right) the task will continue without noticing it was migrated and that it is now running on a different node. Please note that it is assumed that the nodes share a networked file system for the checkpoint file.

Of course, this is only possible if the nodes are identical from the application point of view, so either the nodes should be identical in every way, or some abstraction layer needs to be used. In practice there will always be a layer to take care of differences like the identification of the nodes on the network, even with otherwise identical nodes.

One problem that has recently been looked into is that migrating nodes don't always have a shared file system. In such a case a temporary process is started on the target node which receives the checkpointing image and writes it to the target file system, before continuing to restart the task as if there was a shared file system.

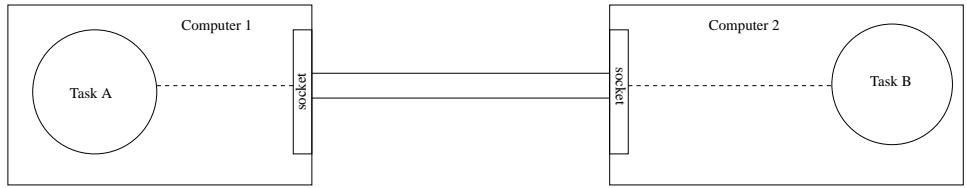


Figure 3.1: Standard communication between tasks

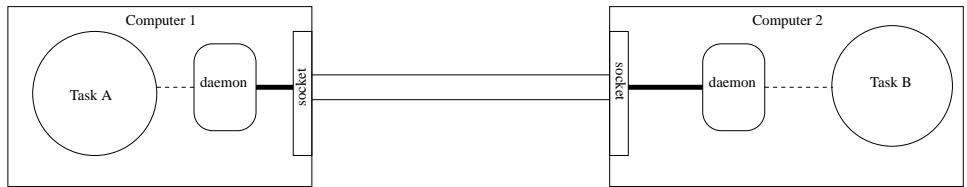


Figure 3.2: Communication between tasks using daemons

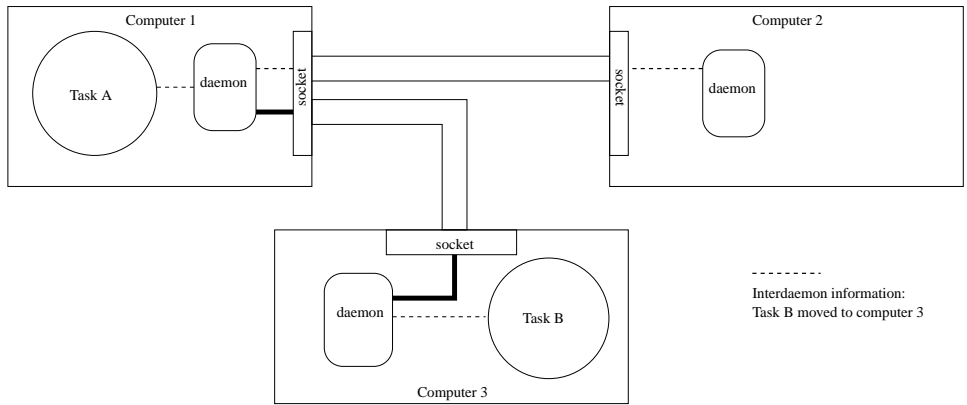


Figure 3.3: Communication with a recently migrated task (in Dynamite)

### 3.3.2 Communication

Communication must not get lost when a task is moved from one node to another. One way of solving this, is to shield off the network subsystem, which can be done in the abstraction layer. Instead of implementing direct communication (figure 3.1), a task communicates with the daemon running on the same node, which will find out which task the packet of information is for (figure 3.2). The daemon will find the node the target task is running on, forward it to the daemon on that node, and the target daemon will forward the packet to the target task.

When a task is migrated off a node, the daemon on that node will remember what node the task was migrated to, and will tell every other daemon that tries to communicate with that task, which node it was transferred to (figure 3.3). This way the sending daemon can update its information about where the task is running, and will send any future messages for the migrated task to the correct node.

The tasks will identify each others location using network-wide unique identifiers, instead of using the actual node identifiers.

### 3.3.3 Open files

Another problem is when tasks have open files, for either reading or writing. First of all, the files need to be available on the target node, with consistent content, and the location where the program was reading or writing needs to be preserved.

Open files can be taken care of by wrapping the file handling calls, again in the abstraction layer. When a task is going to be migrated, the location within each file needs to be stored, the file needs to be closed, the location of the file needs to be noted, and if no shared file system is available, the file itself will have to be transferred to the target node.

At the target node, the file needs to be found, reopened and the file handle set to the correct within the file. All this also goes for directory handles.

## 3.4 Rescheduling

Before migration, a decision should be made what task or tasks are best migrated, and to which node or nodes. This process of deciding the redistribution of a set of tasks is called rescheduling.

The rescheduling problem is similar to the initial scheduling problem. The main difference is that in rescheduling, moving tasks is costly, and information about the behaviour of the tasks is actually expressed in the current network and node utilisations.

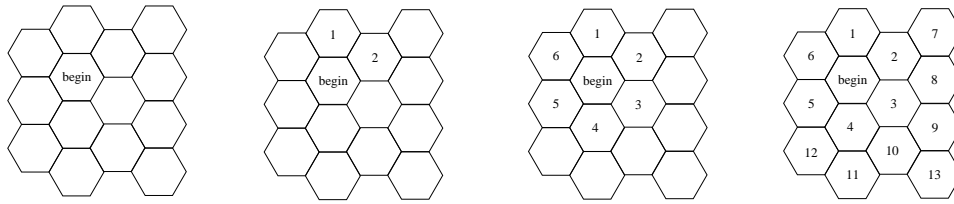


Figure 3.4: Breadth search

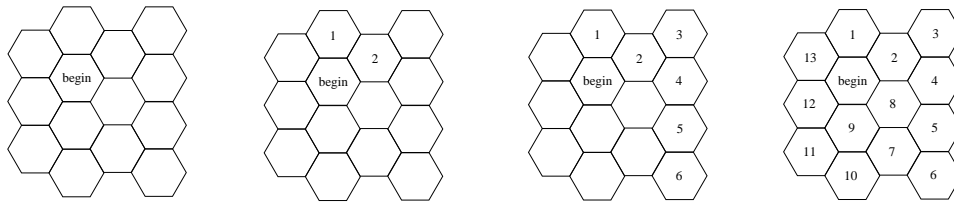


Figure 3.5: Depth search

## 3.5 Search techniques

There are various ways to search for the optimal result in a search space. Since search space is generally not chaotic, but more like a curved surface, with peaks, lows, plateaus and sometimes spikes, one has already some expectation of the value of some point in search space if the value of neighbouring points in search space are known.

The main differentiation in search techniques is between exhaustive techniques, which measure at all specific points in search space, and non-exhaustive which measure at only a subset of all points in order to save time. Exhaustive has the advantage that it will by definition find the very best solution. However, most of the time a good solution will be enough, and the performance difference between that solution and the very best could be very small, even though the solution might be completely different, and only a fraction of the measurement need to be done.

### 3.5.1 Breadth/depth first

Breadth and depth first are exhaustive search algorithms: they try every single solution and simply choose the best.

Breadth first is initially very generalistic. It will try first try all immediate neighbours of the starting point in search space. So, it will vary each of the parameters one by one. It will then move to each of those points and do the same. This is illustrated in figure 3.4: say that neighbours are visited clockwise, starting with the neighbour above. As you can see, first all surrounding hexagons are visited, and after that each of their neighbours,

and so on.

Depth first on the other hand, tries the first neighbour, then the first neighbours' first neighbour and so on. Only when it reaches a point which has no untried neighbours left, will it go back (backtrack) to the previous neighbour to see if it has a second neighbour. In figure 3.5 you can see that the after 6 moves, only 2 of the immediate neighbours of the starting point have been measured.

Because in most problems (including scheduling) a good solution is good enough, and it is not needed to find the very best one, this is rarely used in its pure form.

Instead, the search is stopped after a predefined period of time (or a predefined number of moves). In this situation, the choice between breadth or depth first makes a difference: breadth first will result in a solution relatively close to the starting point, while depth first hasn't tried some close neighbours of the starting point.

Since depth first has the nature to dive into a path that may be very suboptimal, it is possible to create (problem-specific) rules about when it should stop looking in that direction and discard solutions further in that direction. This discarding of entire regions in search space to reduce search time is called pruning. The search is no longer exhaustive, but might find the global optimum nevertheless, depending on how well the pruning rules apply to the problem.

### 3.5.2 Best first/hill climbing

Hill climbing (sometimes referred to by the ambitious sounding name of best first) is a variation on the breadth and depth first search techniques.

It tries all immediate neighbours of the starting position, compares them, and chooses the best one. It tries all immediate neighbours of the best one, choosing the best neighbour again. It continues doing this until all neighbours are worse than the current position. This involves far fewer tries than breadth or depth first would, and is therefore much faster.

For a very large class of problems, a small change in parameters (a single step) will often have a limited effect on the measure that is being optimised. This makes techniques like hill climbing and pruning feasible.

The problem with the hill climbing technique is that it can get stuck if a problem space has multiple locally optimal positions. These locally optimal positions are generally not equally good when compared to one another. Actually, there is no reason to assume that the apparent optimal solution that was found is anywhere near as good as the actual (globally) optimal solution.

While there are a few methods to combat against getting stuck (allowing a couple of steps downhill to worse neighbours for instance), it remains a fundamental problem of best first search.

### 3.5.3 Simulated annealing

Simulated annealing is a search algorithm based on hill climbing combined with a degree of movement of which the scope is based on the time already spent searching. This is done in analogy with the way molecules form crystals when they turn from liquid into solid state: when still hot, molecules have room to move around. As the temperature drops, the molecules are less and less able to move. In this context, temperature is defined as a function of time which gets smaller each time-step.

To start off with, a random location in the search space is chosen. After this, hill climbing is applied. When a random chosen neighbouring state is better, it becomes the current state. If the neighbour isn't better, the move is made anyway with a certain probability, depending on the temperature and how much worse the neighbour is.

With a temperature  $T$  and a difference in performance  $\Delta E$ , the probability could be  $e^{-\Delta E/T}$  [13].

Because of the possibility of descent, local extremes may be avoided, and the relevancy of the choice of the starting point is strongly reduced.

### 3.5.4 Genetic algorithms

Genetic algorithms is the search method I have implemented and will be using for this research.

Genetic algorithms include some randomness, resulting in the ability to jump to a very much different location in a search space with chaotic tendencies. It can also combine two solutions to search for a better one, resulting in a more structured jump.

They will be discussed extensively in chapter 4.



## Chapter 4

# Genetic algorithms

In this chapter I will introduce the concept of genetic algorithms, describe some of the different methods, and how I will be using genetic algorithms for the scheduling problem.

### 4.1 Overview

Genetic algorithms (GA for short) is a optimisation technique from artificial intelligence which is based on the theories on natural evolution by Charles Darwin. The algorithm is as follows:

- Generate a population of chromosomes
- Do until maximum number of generations is reached:
  - Measure fitness of all chromosomes
  - Select fittest for next generation
  - Diversify the population

This is just a framework. All parts of the algorithm can be implemented in a number of ways. The diversification of the population is usually done in two parts: a crossover part which generates new chromosomes based on original chromosomes, and a mutation part where chromosomes are changed without any influence from the population. I will discuss a few of the possible methods for implementing these parts of the genetic algorithm.

### 4.2 Chromosome representation

The problem at hand first needs to be expressed as a chromosome-form. The chromosome is a projection of all properties of a problem at some point in search space. For example, let's take the well known travelling salesman

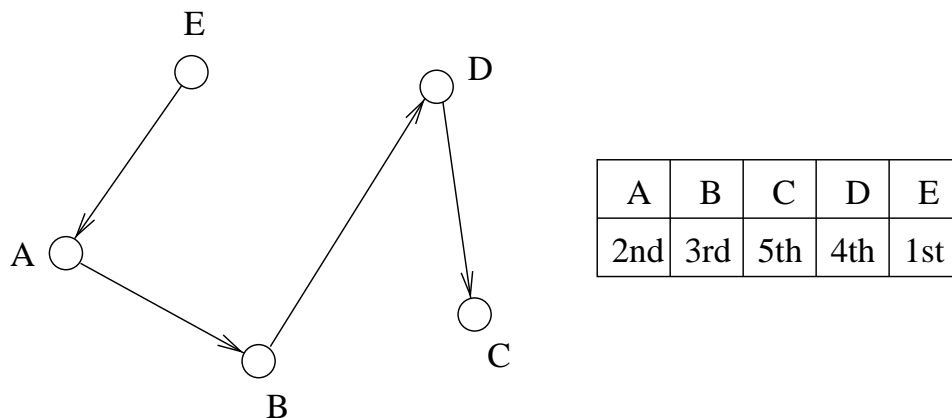


Figure 4.1: TSP example

problem (TSP for short). One possible representation for a single solution as a chromosome would be to make an array of all the cities, and assign sequence numbers for each of them.

For example, when visiting cities A, B, C, D and E, the chromosome  $\{2,3,5,4,1\}$  represents the route in figure 4.1. When a sequence number occurs more than once, simply take the leftmost city with that sequence number first, and so on. This way, all possible chromosomes will represent a valid state.

This representation is somewhat adequate, however it is biased in favour of visiting cities in the front of the chromosome sooner than the cities in the back of the chromosome. This is because the cities in the front of the chromosome have a higher chance of being the first city in the chromosome with a shared sequence number. Therefore, in a running system this would be troublesome, but as an example it will suffice.

A different representation might be to have an single array containing the cities in the order they will be visited. After a diversification, this might yield an array with one city visited twice, and another not visited at all. Therefore, this representation is not usable.

### 4.3 Fitness and selection

Each chromosome has a fitness. This is how well the solution represented by the chromosome fits the problem. So, the optimal solution has the highest fitness, and a very bad solution a very low fitness.

A bunch of chromosomes together constitute a population. Each time step of a population is called a generation. The more fit a chromosome is compared to the other chromosomes in its generation, the higher its chance for surviving into the next generation. Since the population size is kept

constant, a good chromosome is likely to occur more than once in the next generation, while a bad chromosome is likely not to occur at all.

There are various ways to make this selection of chromosomes. To begin with, chromosomes can be chosen like sectors on a roulette wheel. The sectors vary in size, linearly to the fitness of the chromosome they represent. The bad thing about this selection method is that if there is not much difference in the fitness of the chromosomes compared to the absolute fitness value, all chromosomes have a roughly equal chance of being selected to survive. This is even worse when dealing with negative fitnesses.

One solution to this problem is to normalise the fitnesses. One way to do that is by using sigma scaling. With sigma scaling, for each chromosome the difference of its fitness with the mean fitness of its generation is scaled to the standard deviation of the fitnesses. For instance:

$$fitness_{normalised} = \begin{cases} 1 + \frac{fitness - \overline{fitness}}{C\sigma_{fitness}} & \text{if } \sigma_{fitness} \neq 0 \\ 1 & \text{if } \sigma_{fitness} = 0 \end{cases}$$

where C is a constant defining how important the relative fitness is, and  $\sigma_{fitness}$  the standard deviation of the fitness.

Another method for selection is to sort all chromosomes to their fitness, and give a proportional higher chance compared to their ranking. This is called rank selection. For instance with 11 chromosomes, the worst performing might get a chance of 0.5, the second worst 0.6, and so on until the best with 1.5. Other values are of course also possible. The bigger the difference, the bigger the relative penalty and advantage that chromosomes have each selection round.

These selection methods are increasingly computationally intensive. Tournament selection behaves comparable to rank selection while reducing computation [12]. Two chromosomes are chosen from the population, and with some probability  $k$  (higher than 50%) the fitter chromosome is chosen, otherwise the less fit one is chosen.

### Correlation biased selection

One variation on any of these methods, is to scale the fitness based on the correlation with the chromosomes that have already been chosen for the next generation. This is what we have called correlation biased selection, or CBS for short.

If a chromosome is very much the same compared to the chromosomes that have already been chosen for the next generation, it gets a penalty. This way a population has a higher chance of remaining diverse.

Our implementation in combination with tournament selection multiplies

the fitnesses to be compared with the following:

$$1 - \frac{\sum_{i \in I} \sum_{j=0}^L \begin{cases} 1 & \text{if } i_j = x_j \\ 0 & \text{if } i_j \neq x_j \end{cases}}{L \times \text{population size}}$$

where  $I$  is the collection of chromosomes which have already been selected for the new population,  $L$  is the chromosome length,  $i_j$  is the  $j$ th element of chromosome  $i$  and  $x$  is the chromosome under inspection.

### Elitism

Also, you could add functionality to make sure the some top percentage of the generation will never be discarded. This is called elitism [12]. This can be combined with any of the selection methods mentioned above. Elitism is implemented by simply always preserving the best individual in the next generation.

## 4.4 Diversification

After the selection procedure, two procedures of diversification can be applied. There are two main different methods: crossover and mutation.

These are applied with some preset probability. If a chromosome is not chosen for diversification, it survives into the next generation unchanged.

### Crossover methods

Crossover is the combination two chromosomes. This is in its simplest form done by cutting the two chromosomes at some, for both chromosomes the same, random location in the chromosome, combining the first part of the first chromosome with the second part of the second, and the first part of the second chromosome with the second part of the first. For instance,  $\{2,5,4,1,3\}$  crossed with  $\{3,4,5,1,2\}$  after the second element would result in  $\{2,5,5,1,2\}$  and  $\{3,4,4,1,3\}$  as offspring if crossed at the second gene.

Other crossover methods could be uniform crossover [14, 4], which decides for each separate element if it is taken from the first parent or from the other. The uniform crossover of  $\{2,5,4,1,3\}$  and  $\{3,4,5,1,2\}$  might result in  $\{2,4,4,1,2\}$  and  $\{3,5,5,1,3\}$ .

### Mutation methods

With simple mutation, each gene in a chromosome has some chance (set by the mutation probability parameter) of changing to a random valid value.

With swap mutation, the contents of two genes are interchanged. In the case of scheduling, this is the same as two tasks swapping nodes. However,

no new values will appear in the chromosomes, only existing values will be moved.

An additional method for mutation, specific to scheduling, is to scale the chance nodes are chosen to their respective processor speeds. For instance, a slow node will have only half the chance of receiving the additional task than a faster node.

## 4.5 Design choices

There are two things that have to be done right for Genetic Algorithms to work. Firstly, the representation needs to be done in such a way that all possible chromosomes are correct solutions, and that all solutions can be represented. The representation should not be biased toward some solutions over others.

The second is the fitness function. While assigning a number for the quality of a solution is in general not a very complicated thing to implement, it does have to be taken into account that the fitness function will be called for each chromosome in every generation. Because this will make up for most the processing time, this should be implemented as efficiently as possible.

In the representation which I decided to use for the scheduling problem, each element of the chromosome represents a single task, and it contains the node on which it runs. Example, in the case where a schedule consists of tasks A and C running on node 1 and task B running on node 2, the chromosome is {1,2,1}. This is a correct and mainly unbiased representation.

If, for instance, I had chosen the units to represent a certain amount 'task slots' per node, of which each slot may or may not contain a task, the number of tasks per node be limited to the number of slots. Furthermore, after migration or crossover, illegal strings could be formed because a task might not be present, or multiple times, in the solution.

The first representation is a very usable one because all crossed and/or mutated chromosomes are valid as a schedule, all possible schedules can be represented, and the representation is not biased to one schedule more than to another.

The fitness function is more problematic. It is impossible to use a real system for this, because of the cost of a single try. Because of the way a parallel system works, it needs some time to be set up, a while to be running consistently, and some time to be shut down. Since the fitness function is called *population size \* the number of generations* times, this would require a gigantic amount of machine power. Not something one would like to do when the objective is to enhance performance.

A better solution is to model the parallel system so tests can be done without running an actual parallel system. Of course, this model will have to be validated against a real system in order to yield correct results.

Such a model was made available to me in the form of `perf-model`, and I will adapt the model so that it can serve as my fitness function.

## Chapter 5

# Performance model

In this chapter I will describe the performance model made by Dick van Albada that I will be using for my experiments. This model is as yet not validated against a real cluster. Validation is therefore part of the remaining work.

The model consists of

- the definition of the elements of the cluster itself - cluster model
- a measure for how work is done, in what amount - execution model and work measure
- the specific clusters and programs that are being modelled.

Additionally, some optimisation methods that have been implemented will be described.

### 5.1 Cluster model

The performance model represents the entire scope of how a cluster of nodes can run parallel programs. It has three different sections: a physical section, a communication section and a program section. A few examples will be given in section 5.4.

The physical section consists of hubs, lines and nodes. The hubs connect different nodes and other hubs to each other through lines. Both the lines and the hubs have a utilisation, an amount of traffic, a latency and a limited capacity, while a node has a network speed and a network usage. All these factors have different effects on the network speed between all the different nodes. A node of course also has a CPU speed, a usage and a number of CPUs.

The communication section has lines, routes, communication links and channels. The lines represent the physical wires, either a node or a hub is connected to each end. The routes are the path of lines and hubs that

messages follow when they are sent from one node to another. They forward the messages from a hub to another hub that is closer to the target node.

Channels connect tasks together, and consist of one or more communication links. Communication links are either a node or hub with its line to the next hub in the channel, or the final node. Which communication links form a channel depends on the routes of the nodes which the tasks are running on.

A program section has tasks, programs and schedules. A program consists of one or more tasks, of which one is limiting execution rate of the program. A task is running on a single CPU and can have multiple channels in order to be able to communicate with other tasks.

All programs together, including all the extra information about the hubs, nodes and routes, form the complete schedule.

## 5.2 Work measure

The measure of how much work is done in a process is calculated in four different parts. The execution speed is limited by:

1. the amount of available CPU time,
2. the amount of available communication capacity on the node the process is running on,
3. the communication channels are limited by the speed of the hubs and lines, and
4. the communication/calculation balance in tasks and delays because of task switching.

The limitation of the execution rate is done in the same way in all four cases.

For instance, with CPU usage, all tasks have an amount of CPU time they request. If all these requests together are less than the combined speed of the CPUs of that node, then all requests can be granted (with the exception that a task cannot be assigned more than 1 CPU at a time, and therefore cannot use more than 1 CPU worth of processing time).

If not, then all tasks receive a slice of CPU time: the total available CPU time is divided by the number of tasks (again, with the maximum of 1 CPU worth of CPU time). Some tasks do not require their entire slice, and the remainder of their slice will be used by the tasks that do.

After all this, each program is evaluated to see which task is the limiting task of the program, and what the performance penalty of that tasks on the other tasks is.

The amount of work that is done in some time step, times the priority of that program relative to other programs, is what is used as a measure for the amount of work that is done given the current schedule, as expressed in the following formula:

$$\sum_p \textit{priority}_p \times \textit{work}_p$$

where  $\textit{priority}_p$  is the priority of program  $p$ , and  $\textit{work}_p$  the amount of work that has been done concerning program  $p$ .

## 5.3 Optimisation methods

The model uses a few different optimisations. I will discuss the most important ones here.

### 5.3.1 optPerProgramme

optPerProgramme, as the name implies, optimises a schedule on a per-program basis.

From `optimise.c` :

optPerProgramme uses some heuristics to find likely candidate tasks to move, and moving them to likely target nodes first. The tasks are found by first looking at the highest priority programme, finding the limiting task for that programme, and then trying to move one task from the host node of that task to another node. All destination nodes are examined in order of "elasticity". For now, the sorting of the destination nodes is not important, as all nodes are examined. When a better schedule is found by so moving a task, optPerProgramme returns immediately. When no better schedule can be found, all nodes and tasks are still examined exhaustively.

The elasticity is a measure of how much the situation of the node changes when a task is added or removed. It shrinks logarithmically smaller to 1 as the number of tasks running on it increases. It is defined as follows:

$$\left(1 + \frac{1}{\# \textit{tasks} + 1} - \textit{Utilisation}\right) \times \# \textit{CPUs} \times \textit{Speed}$$

Also, tasks from other programs on the nodes with program-limiting tasks are also considered for migration. This entire process is looped until no further enhancements can be made.

### 5.3.2 lessStupidOpt

lessStupidOpt is a simple migration algorithm. It sorts all nodes, starting with those which have many tasks relative to their speed, and ending with those with few. Then it tries to move each task, one at a time, from the current node to another node, trying nodes with large elasticity first. As with optPerProgramme, this process is repeated until no further improvements are obtained.

### 5.3.3 doubleOpt

doubleOpt is a hybrid of lessStupidOpt and optPerProgramme. Initially, it works like lessStupidOpt, however it checks only the three heaviest loaded nodes instead of all. Additionally, if no change can be made in these three in order to gain performance, optPerProgramme is used to optimise further.

Since optPerProgramme works on a per-programme basis while lessStupidOpt works on a per-node basis, this might result in a different solution than the solution found after the lessStupidOpt part. If this is the case, doubleOpt is restarted so that further improvements can be made with the current solution as starting point. This is again done until no further improvements can be made.

## 5.4 Modelled clusters

In this section I will describe the clusters and programs that I have used for my experiments.

### 5.4.1 test-input7b

I will start with the case that has been modelled that I have been using the most, both in the experiments and in debugging the genetic algorithm. The hardware of the modelled cluster in question is illustrated in figure 5.1.

The circles represent the hubs, and the squares represent nodes. The thickness of the circles represents the maximum amount of data a hub can handle in an certain amount of time (the capacity), as the thickness of the lines represent how much traffic the lines can handle.

The numbers in the circles are hub reference numbers. As is illustrated, all the latencies of the lines connecting the hubs is equal, except for the one between hubs 0 and 3, which is longer. The latencies of the hubs themselves are equal, and therefore not illustrated.

The size of the squares represents the speed of the nodes. The nodes directly connected to hubs 2, 3 and 4 are twice as fast as those connected to hubs 0 and 1, and their connection to the hubs is also twice as fast

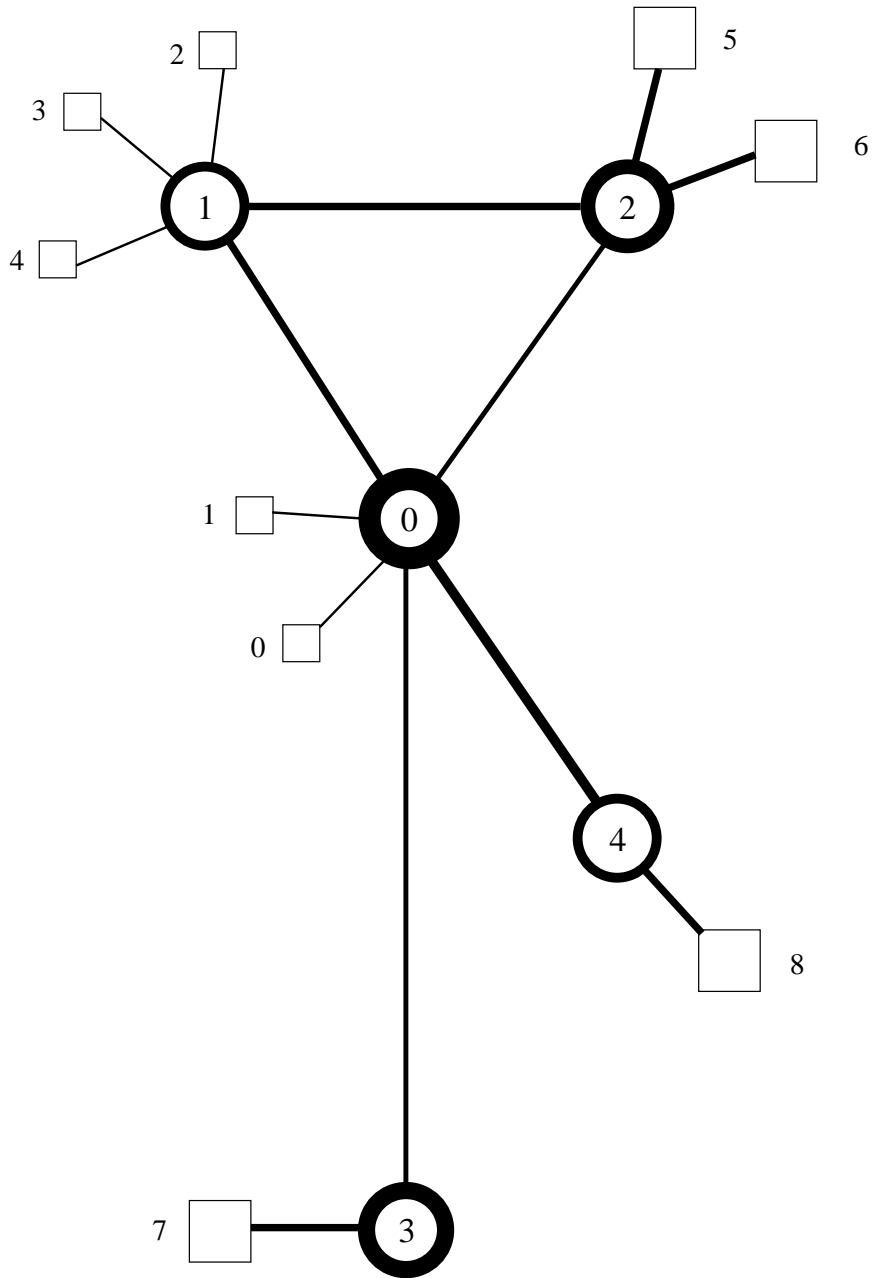


Figure 5.1: Hardware in model test-input7b

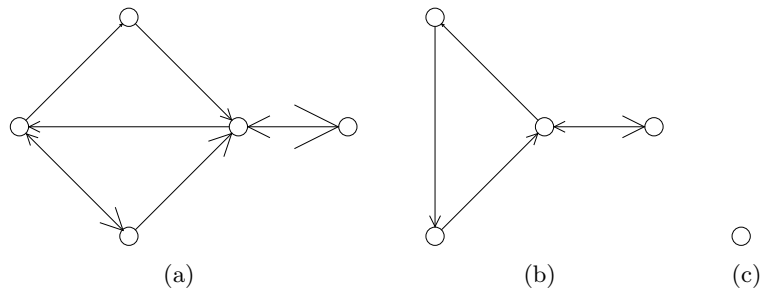


Figure 5.2: Software in model test-input7b

(illustrated by the thickness of the lines). The numbers by the squares are node reference numbers. All nodes have a single processor.

The programs that are modelled in test-input7b are shown in figure 5.2. Each circle represents a task within the program. As you can see, one of the programs is a serial program (figure 5.2(c)): it consists of a single task and therefore has no communication channels.

The amount of communication is represented by the size of the arrow-heads. For instance, in the program in figure 5.2(b) you can barely see the arrow between the middle task and the uppermost one, while there is a lot of communication between the middle task and the right one, illustrated by the large arrow head.

What is not represented in the graph, is how much the tasks are dependant of the others.

In test-input7b, the left program appears once, the middle one twice and there are 4 serial programs. The two middle copies of the program illustrated in 5.2(b) differ in inter-task dependency only.

#### 5.4.2 test-input9

Test-input9 has many similarities with test-input7b.

In hardware, nodes 0, 1 and 8 in figure 5.1 are changed to dual-processor machines, and a copy of the dual-processing node 8 is added to hub 0, becoming node 9. Except for those differences, the hardware stays the same.

Compared program-wise, test-input9 also has 4 serial programs, and twice the smaller parallel program from test-input7b (figure 5.2(b)). The program in figure 5.2(a) has been replaced by the program shown in figure 5.3.

In this new program, all 8 tasks have a communication channel with the next task, with the last task connecting to the first. You might say they form a communication ring. Also, they all have a communication channel with the 4th task ahead, which is the exact opposite task on the ring. This

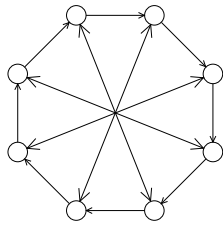


Figure 5.3: test-input9

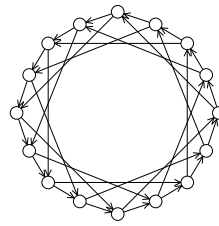


Figure 5.4: test-input16d

communication channel carries twice as much information as you can see by the size of the arrowheads.

### 5.4.3 test-input16d

The hardware in test-input16d is simply 16 equal nodes connected to a single hub. This is the situation one would expect in a dedicated cluster.

The program in test-input16d can be seen in figure 5.4. It is slightly based on the larger problem in test-input9: every task has a connection to the next, and a connection to four tasks ahead. However, unlike the program in test-input9, the two communication channels are of the same speed, and four tasks ahead does not result in an bidirectional diagonal channels.

The purpose of this particular modelled cluster is to test in how far local flats can be escaped.

Since the number of tasks is equal to the number of CPUs, every task on a different CPU is a special case. In this case all the communication needs to be passed over the line, possibly congesting the network, while the calculation part of each task will be as fast as is possible with this cluster.

Another special case would be to always put two adjacent tasks on a single node, eliminating a quarter of the networked communication. However, if for instance tasks 1 and 2 are running on node A and tasks 4 and 5 are running on node B, task 3 will be unable to couple with an adjacent task unless in other tasks is made. This might temporary result in a worse performing solution and is therefore a good example of a local maximum.



# Chapter 6

## Results

This chapter will discuss the different approaches for the genetic algorithm I have chosen to implement and run experiments for, and the results of the experiments.

### 6.1 Different approaches

In genetic algorithms, there are a couple of parameters that can be adjusted in order to try to yield better results. These parameters are:

- crossover probability,
- mutation probability,
- population size,
- maximum number of generations, and
- tournament parameter  $k$  (if tournament selection is used).

The time the genetic algorithm takes is proportional to the population size times the number of generations. This is because the fitness function is by far the most time consuming part of the algorithm.

Because time is limited in scheduling and therefore not very flexible, population size and the maximum number of generations are inverse proportional. Therefore, either the population size or the maximum number of generations can no longer be considered a parameter, as it is set by the choice for the other. I have chosen the population size to be the parameter to be set. This way, the time that the genetic algorithm may take becomes a constant.

Beside these parameters, there are a few different approaches for the various steps to be taken in a genetic algorithm. We consider:

- roulette wheel selection versus tournament selection,

- straight chances in selection versus elitism and/or correlation biased selection,
- simple mutation versus swap mutation,
- straight chances mutation versus speed-scaled chances mutation, and
- simple crossover versus uniform crossover.

## 6.2 Preliminary setting of parameters

Since genetic algorithms are non-deterministic, multiple measurements for each set of parameters need to be made in order to have a reliable value for what would be expected in practice. I have done each experiment either 60 or 100 times and have used the average for the results shown here. We begin with the following parameters:

- test case: test-input7b
- crossover probability: 0.4<sup>1</sup>
- mutation probability: 0.1<sup>2</sup>
- population size: 20
- maximum number of generations: 1000
- tournament parameter: 0.75 (where applicable)
- number of test averaged for each graph: 60
- roulette wheel selection
- no correlation biased selection
- no elitism
- simple crossover
- simple mutation
- not CPU-scaled mutation

For each test, the fitness of the best chromosome encountered thus far is kept (outside the population), and the average of those best chromosomes is drawn in the figures as a function of the number of generations.

---

<sup>1</sup>Each chromosome has a 40% chance of being selected for crossover.

<sup>2</sup>Each gene has 10% chance of being selected for mutation.

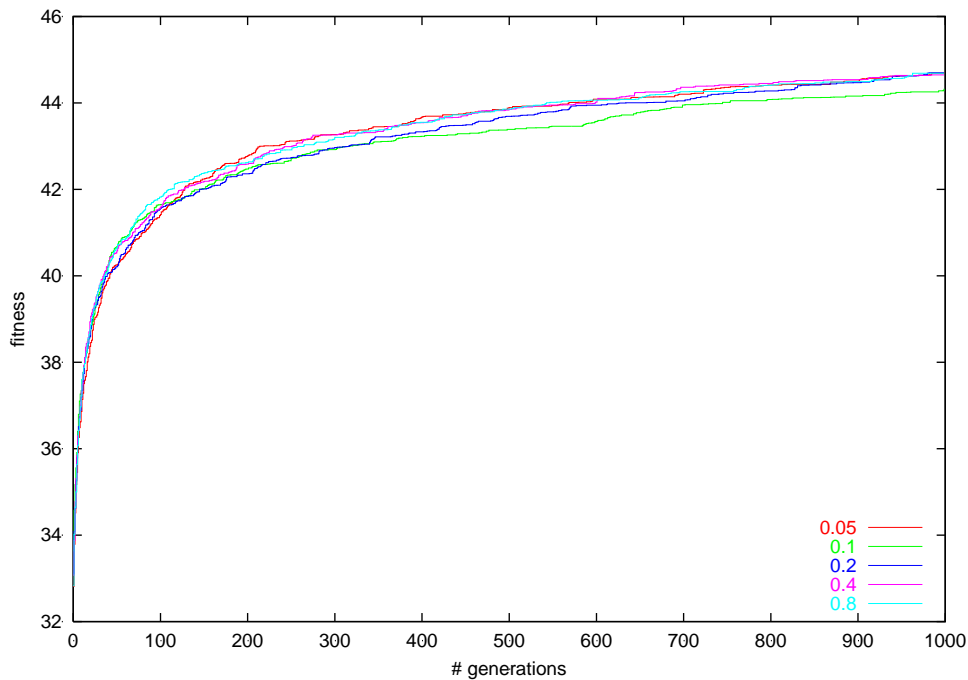


Figure 6.1: Crossover probability comparison

The first thing that needs to be done is to have some impression of what the crossover and mutation probabilities and population size (together with the maximum number of generations) should be. This should not become too problem-specific, since we'd like to keep the same parameters usable for other problems. It will be further refined later on. Let's begin with crossover. The results are shown in figure 6.1.

This graph is pretty unclear. Since this was also the case for later experiments not shown here, the results will be shown with the number of generations on a logarithmic scale, which provides more clarity.

As can be seen in figure 6.2, the crossover probability doesn't seem to have much effect. We'll come back to this later, the crossover probability will be kept at 0.4 for now.

In figure 6.3, different mutation probabilities are compared. These results are more meaningful: the lower the mutation probability, the lower the fitness is in the beginning, but the higher it becomes after a while. The optimal value will therefore depend mostly on the amount of time available for computing a schedule. However, 0.05 seems to be quite a good value overall until about 120 generations where 0.025 takes over. For the next experiment a mutation probability of 0.05 will be used.

In figure 6.4 the results of different population sizes is shown. In this graph the number of generations is no longer a good cost measure because

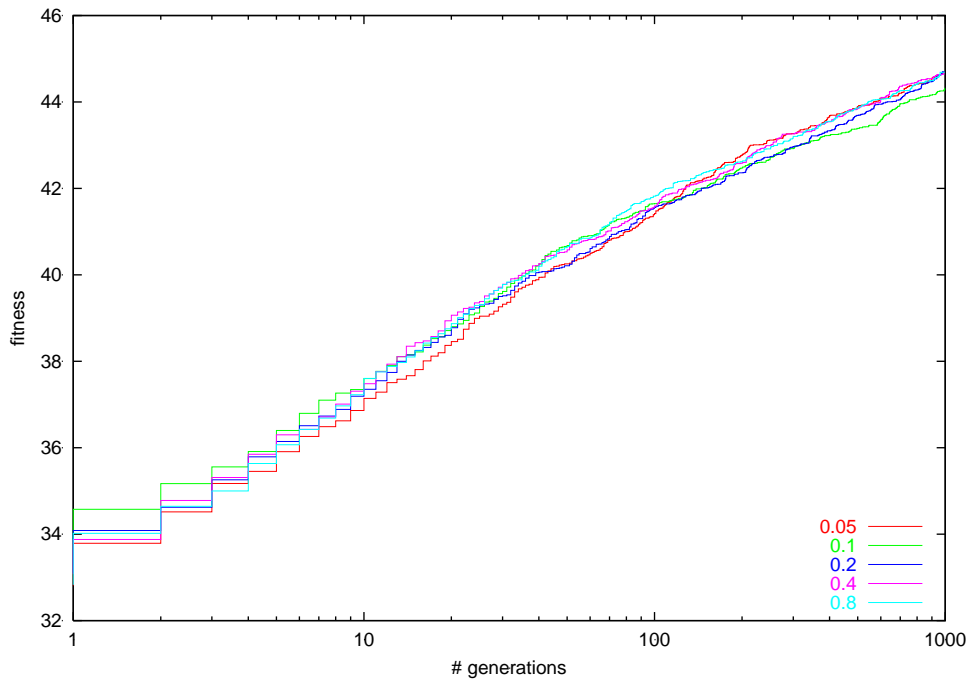


Figure 6.2: Crossover probability comparison, logarithmic

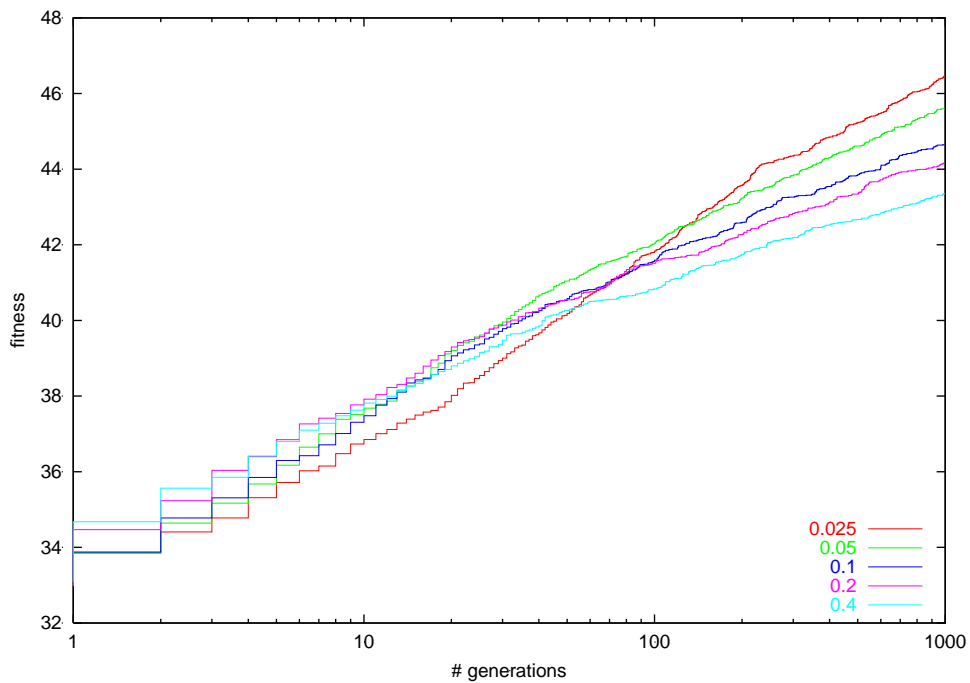


Figure 6.3: Mutation probability comparison

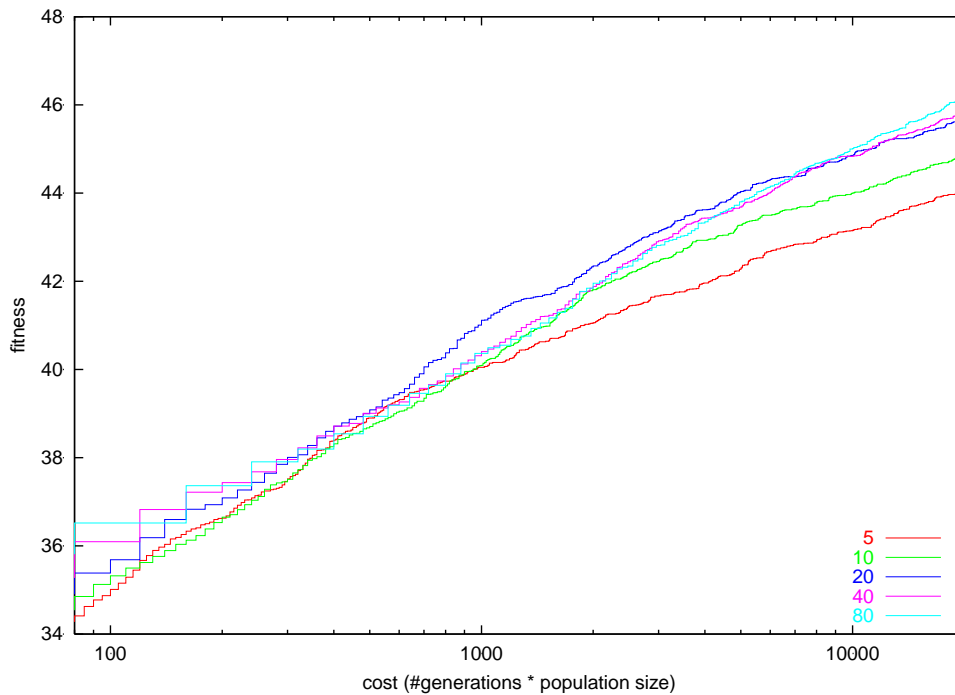


Figure 6.4: Generations/population size balance comparison

a smaller population needs linearly less time for fitness testing.

As with mutation, we see a balance between the parameter and the time that is being put into the experiment. Although 5 and 10 are clearly worse than the others, 20 can also be seen to be worsening at the end of the graph. However, since the maximum number of runs was by this time already clearly taking too much time, and 20 performs better in the 600 until 6000 fitness tests-region, 20 is chosen for population size.

We now have some preliminary parameter settings. We will come back to these settings later to check for dependency.

### 6.3 Selection methods

Now that we have some basic parameter settings for the genetic algorithm, it is time to see what difference alternate methods of the algorithm can accomplish.

In figure 6.5, the results of the different selection methods is shown. Independent of what further selection is used, elitism makes a big difference. Any selection method with elitism enabled is performing better than any selection method without. Therefore, elitism will be used in all experiments from now on.

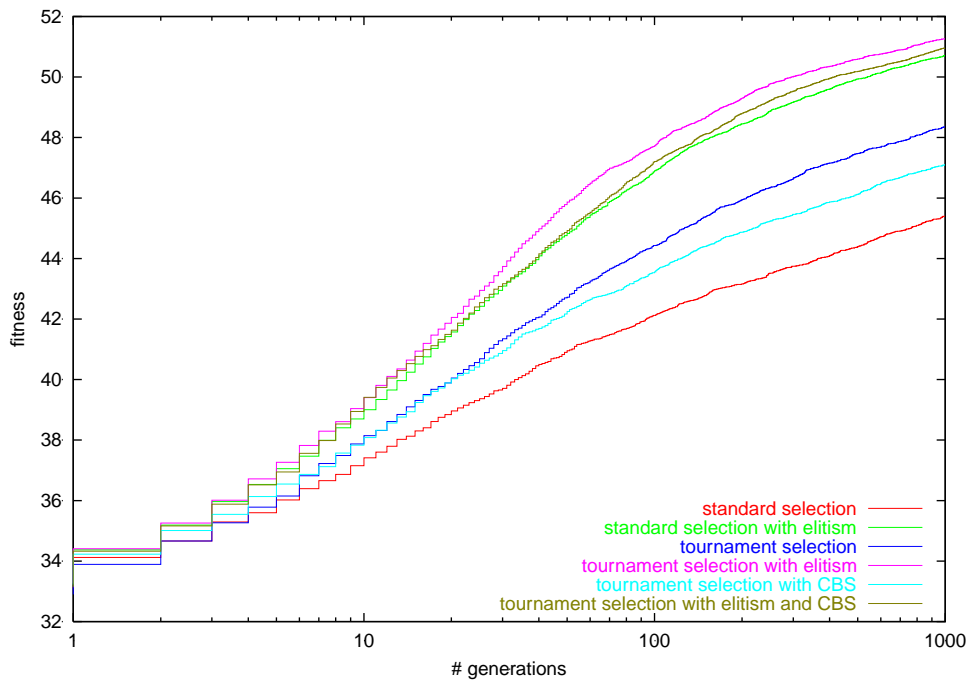


Figure 6.5: Selection method comparison

Furthermore, the tournament selection in performs better than the roulette wheel selection, either with or without correlation bias. Therefore, tournament selection was used for further experiments.

Apart from this, the correlation biased selection that makes sure that a population stays diverse, seems to have a somewhat better results than the standard roulette wheel selection, but not as good as the tournament selection without the correlation biased selection.

Tournament selection introduces a new parameter: the tournament parameter. The results of the measurements to set this parameter are shown in figure 6.6. The results from the roulette wheel selection are shown for reference.

It is again a balance between the parameter and the time that is available. After some more experiments for the tournament parameter (which are not shown here because they would further complicate the graph), 0.98 was chosen as the optimal setting.

## 6.4 Different models

Now it is time to look at the other cases as well: test-input9 and test-input16d. Figure 6.7 is the updated version for the generations/population

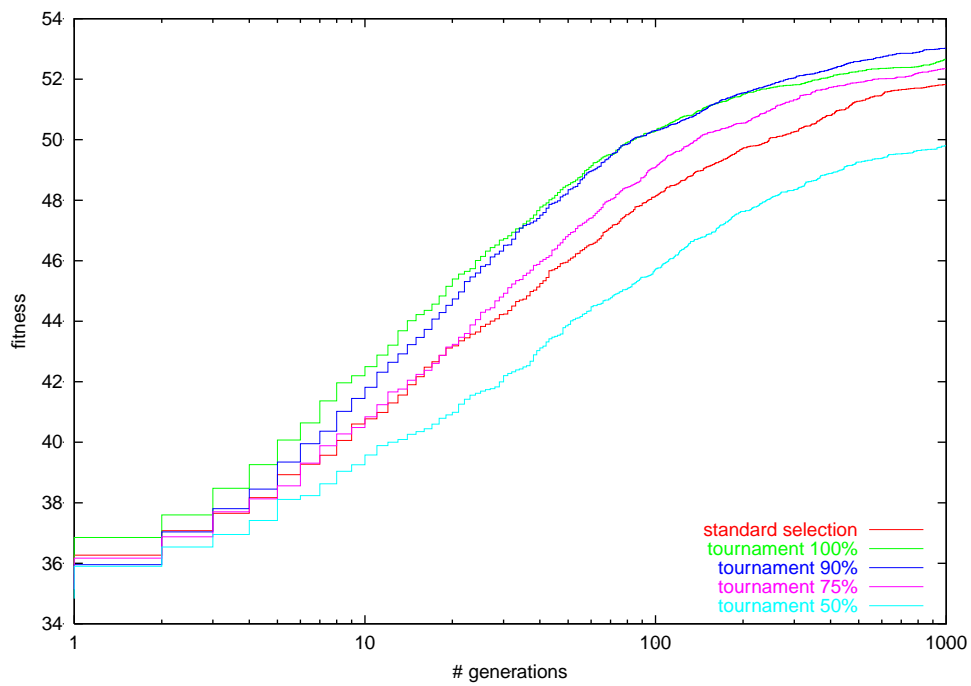


Figure 6.6: Only the strong survive?

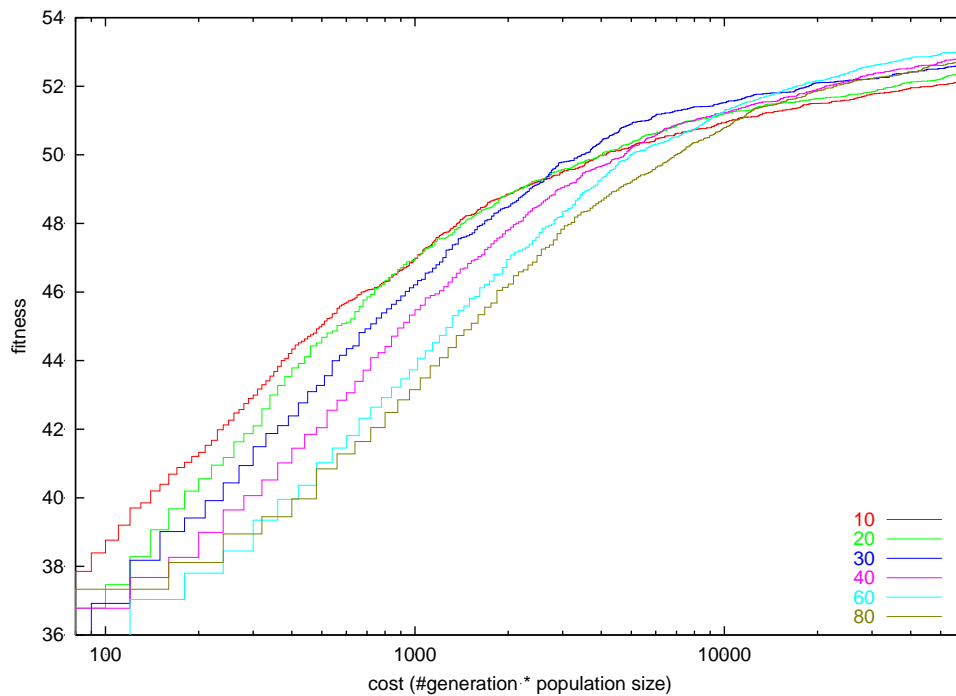


Figure 6.7: Generations/population size balance, test-input7b

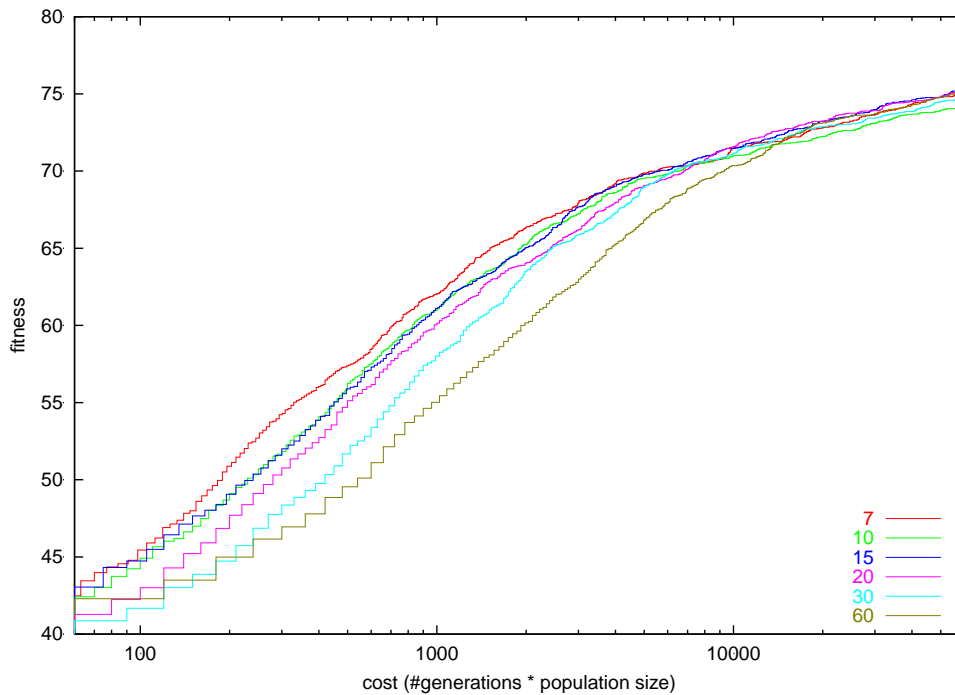


Figure 6.8: Generations/population size balance, test-input9

size balance for 7b as we have already seen in figure 6.4, only now with the tournament selection and elitism.

Figure 6.8 shows that after some 12000 fitness tests, the population size has hardly any effect when optimising test-input9.

Test-input16d is a test case where there are many equally good solutions, which frustrates the genetic algorithm as can be seen by the noisiness of figure 6.9. The standard deviation of the tests is much larger with 16d than with 7b or 9: up to 15 versus about 2 and between 3 and 4 respectively, as can be observed in figure 6.10. The standard deviation remained fairly constant in following experiments.

In this case, when using an error measure of

$$error = \frac{\sigma}{\sqrt{n}}$$

where  $n$  is the number of tests measured, the error in fitness is almost 2, making the differences nearly insignificant. However, from 2000 up to 15000 runs a population size of 20 is slightly better than the others.

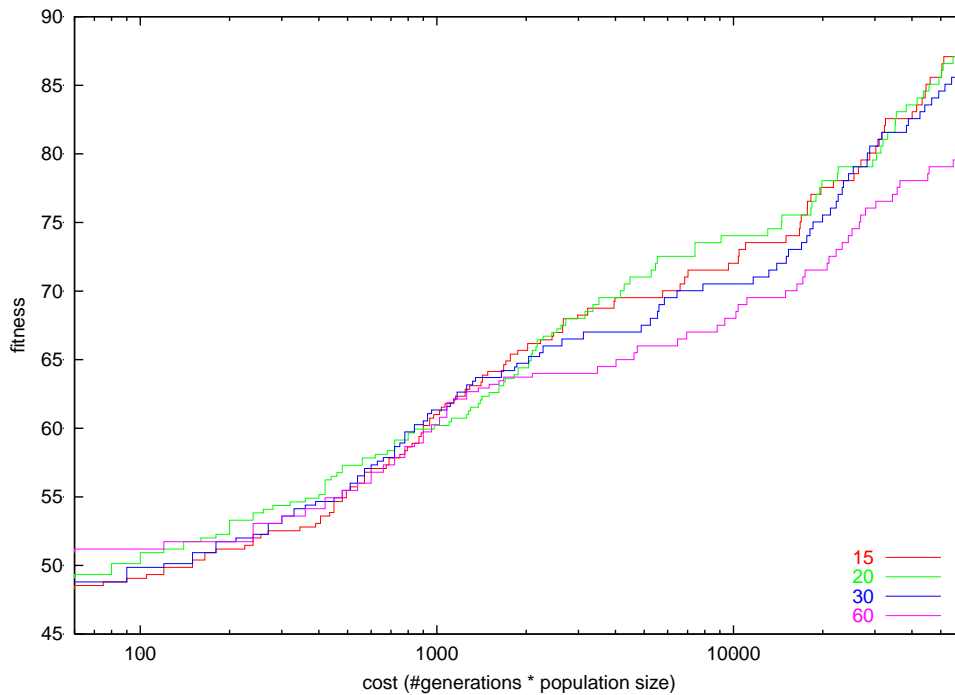


Figure 6.9: Generations/population size balance, test-input16d

## 6.5 Number of generations/population size balance revisited

At this point in the research it became problematic that a fitness run of one test case might take more time than the fitness run of another, and therefore the previous way cost was represented (number of generations times population size) is no longer useful.

To combat this problem, a system was implemented in order to make each run of the genetic algorithm for some set amount of time. Of course, this time needs to be a realistic measure for the time that can be taken in a real-life situation.

The choice for this parameter has been chosen as 3 seconds of CPU-time on a Pentium III 1 GHz machine. It finishes the generation it is working on 3 seconds after it has started, and returns the best individual which was encountered. This way, we can more easily compare the results of different test cases to see what the overall best choices for the parameters are.

In figure 6.11 the fitness for test-input7b is plotted as a function of the crossover and mutation probabilities given a population size of 20 and the tournament parameter at 0.98, with tournament selection, elitism, simple crossover and simple mutation.

The contour lines at the bottom of the graph give a further indication of

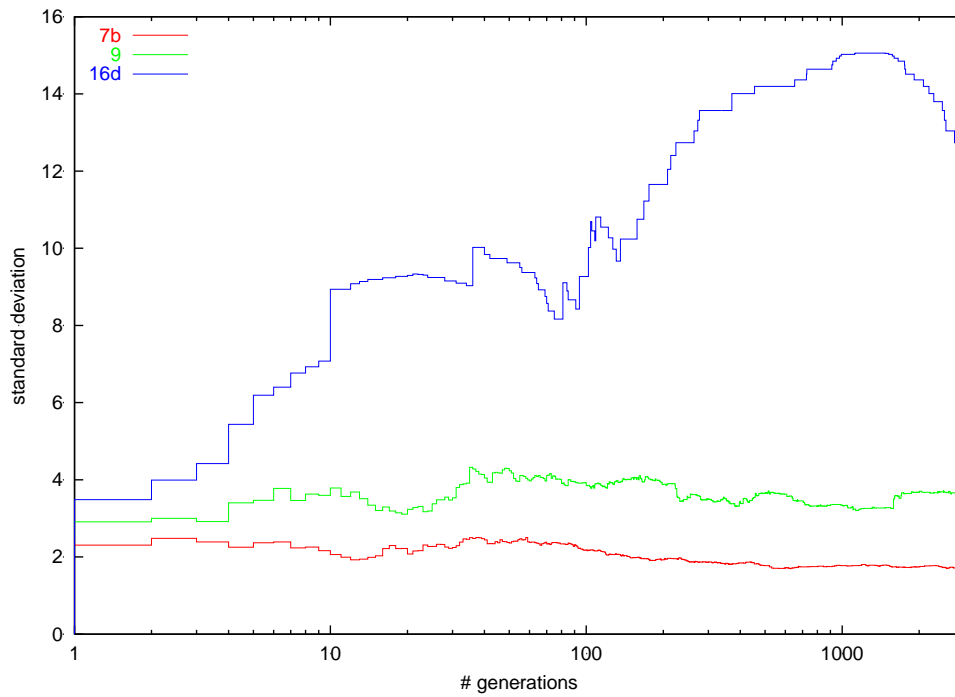


Figure 6.10: Typical standard deviations for the three problems (population size: 20)

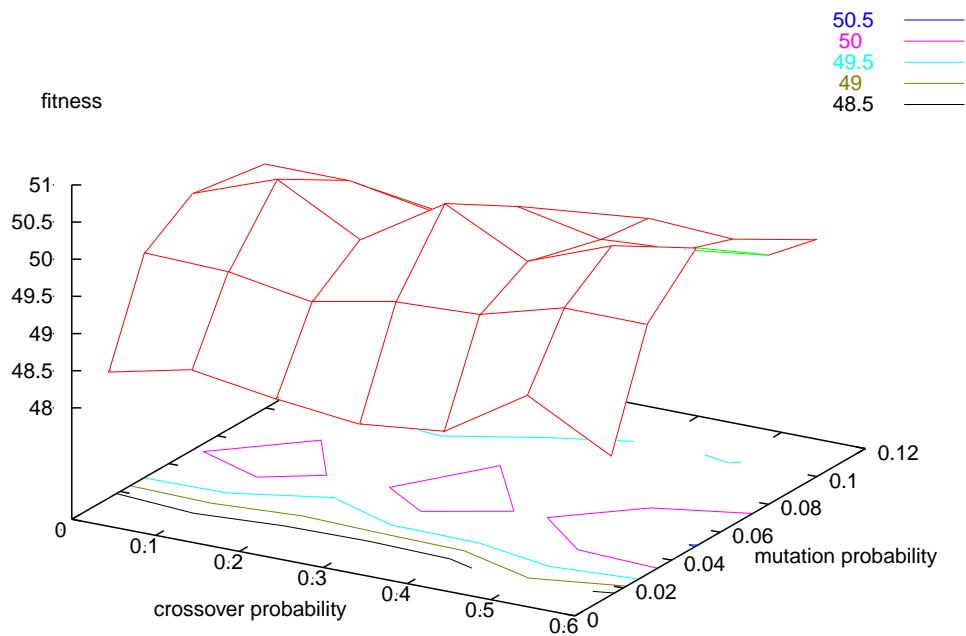


Figure 6.11: Crossover/mutation dependency, test-input 7b

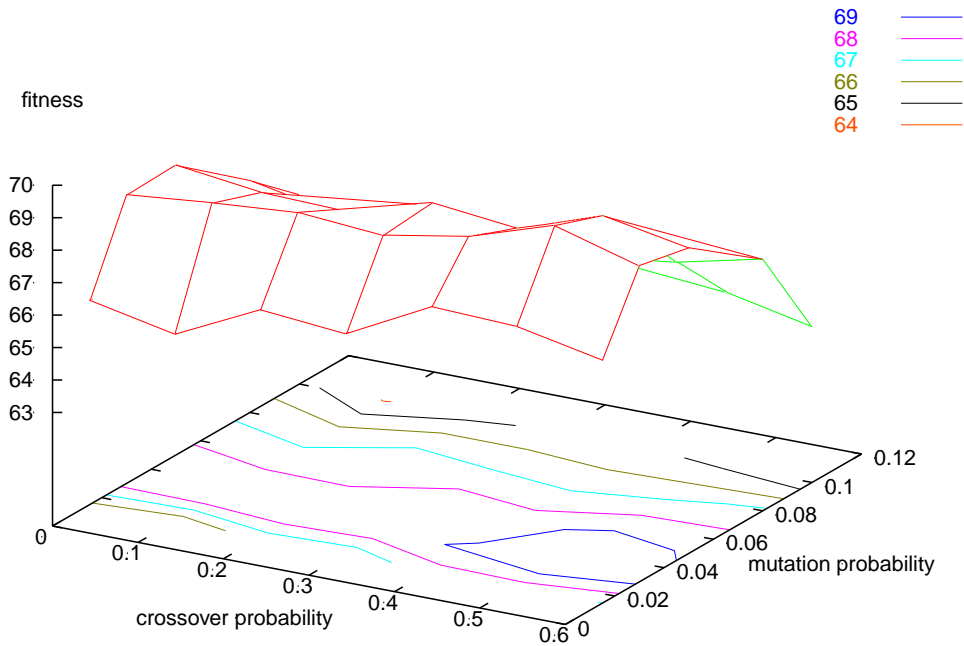


Figure 6.12: Crossover/mutation dependency, test-input9

the fitness values, and make it more easy to compare specific points in the plots.

As you can see, the situation since the first crossover experiments hasn't changed much under the influence of the different selection methods and such: the crossover still doesn't have a large influence. Even when the crossover is disabled (with the probability at 0.0), not much of a difference can be observed. The mutation parameter makes more difference, and the best results are obtained if the mutation probability is set at about 0.05.

In figure 6.12 the same is done for test-input9. Here the crossover seems to have a small influence: at 0.5 it has a slightly better result than with other values. The mutation yields optimal results with a probability of 0.03.

Test-input16d (figure 6.13) is somewhat more erratic. The best results are when using crossover probability 0.2 and 0.5 with mutation probability 0.03.

## 6.6 Alternate differentiation methods

In this section the results will be shown for the alternate methods for mutation and crossover.

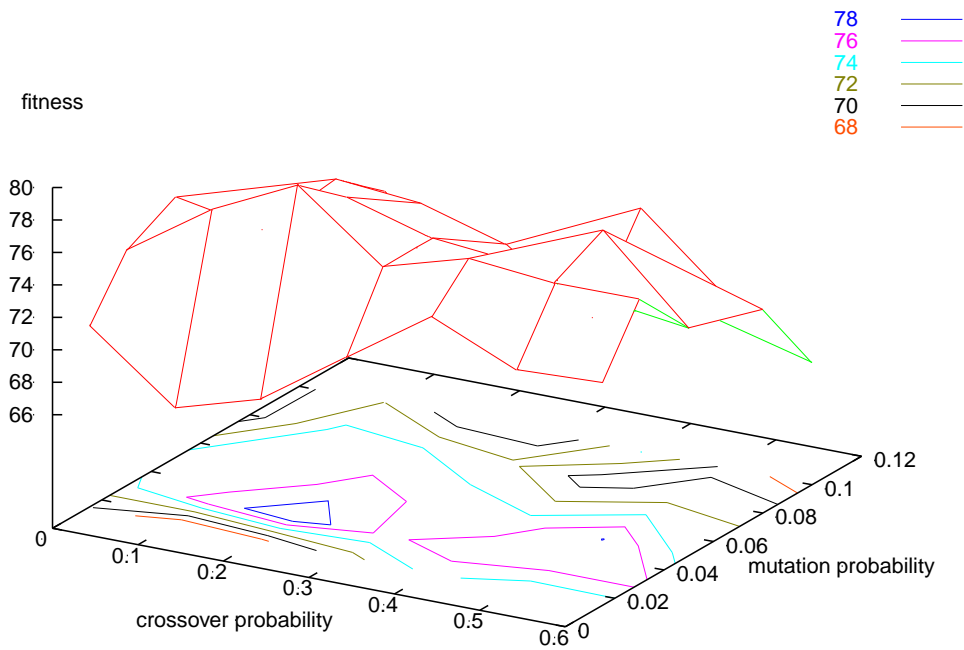


Figure 6.13: Crossover/mutation dependency, test-input16d

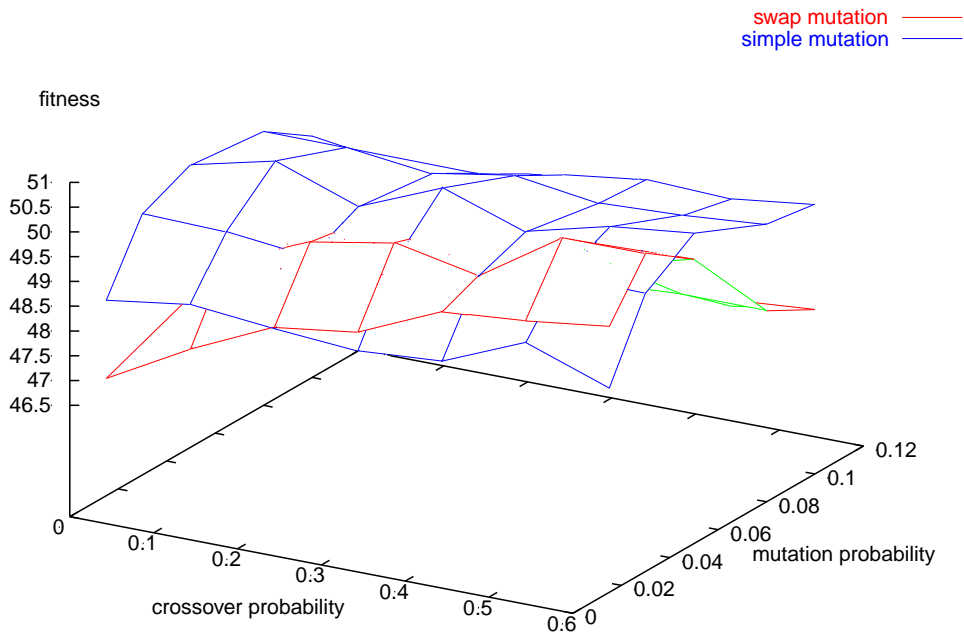


Figure 6.14: Normal versus swap mutation, test-input7b

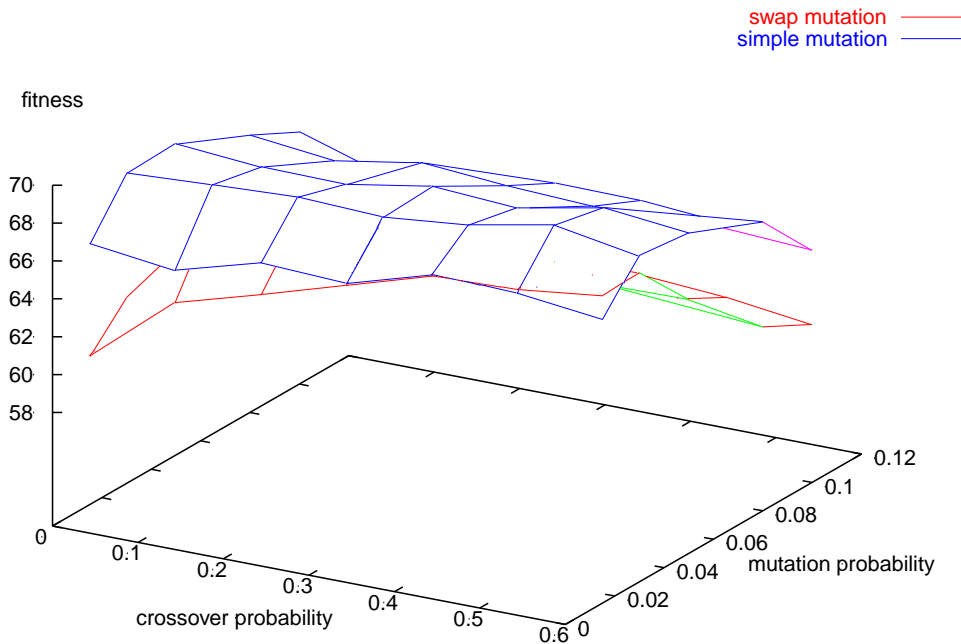


Figure 6.15: Normal versus swap mutation, test-input9

### 6.6.1 Swap mutation

In figure 6.14, swap mutation and simple mutation are compared. Swap mutation has a bad effect in most cases. But in a few cases it did generate a higher fitness than the simple mutation, and it also has the highest fitness overall with a crossover probability of 0.5 and a mutation probability of 0.02.

As can be observed in figure 6.15, with test-input9 swap mutation is rarely a better solution than simple mutation. It also doesn't have the highest fitness overall as with test-input7b.

In figure 6.16 the problem of swap mutation can be observed as it has no longer the ability to move a single task. Also, no as yet unused nodes can be introduced, while nodes may disappear from the entire population during the selection procedure. This is a fundamental problem of using swap mutation. A solution might be to use both swap and simple mutation, which was not tested in this research.

### 6.6.2 Scaled mutation

In figure 6.17 it is shown that test-input7b benefits somewhat from the higher chance of being migrated to a fast node when a mutation occurs. The overall peaks are also achieved by the scaled mutation. Test-input9 has comparable results.

With test-input16d (see figure 6.18) it is once more erratic. Straight and

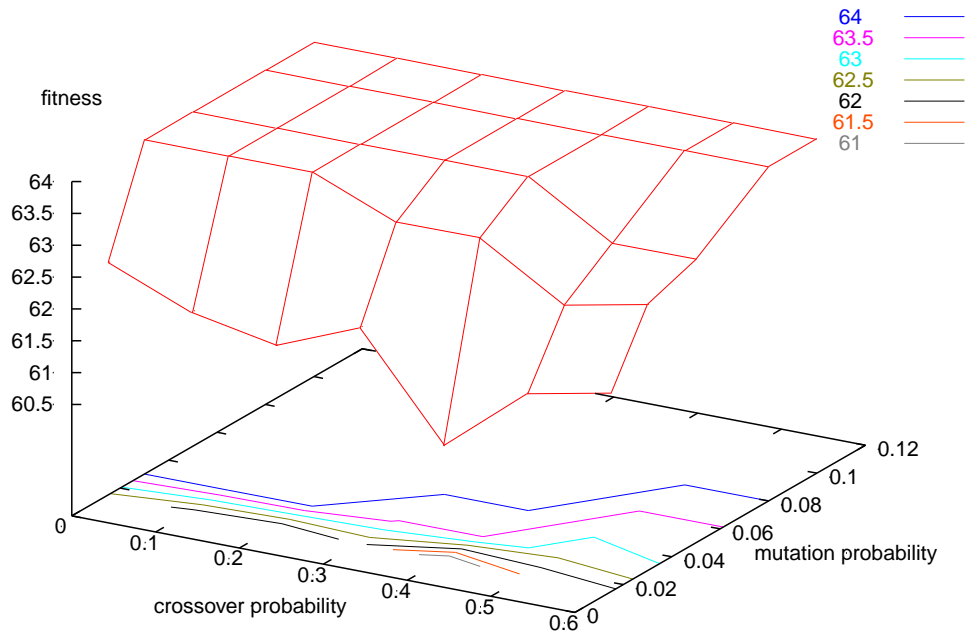


Figure 6.16: Swap mutation plateau with test-input16d

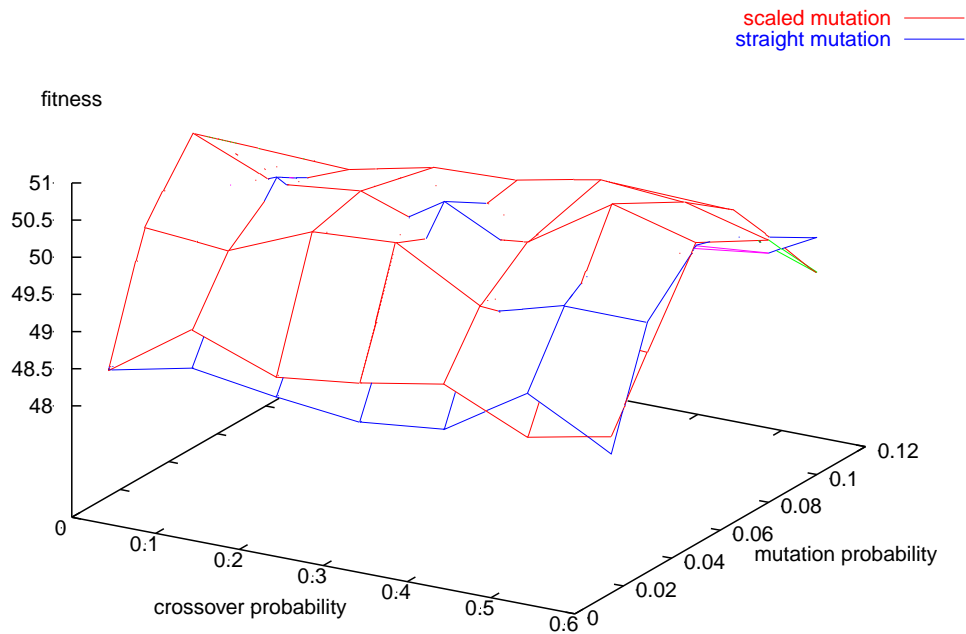


Figure 6.17: Straight versus speed-scaled mutation, test-input7b

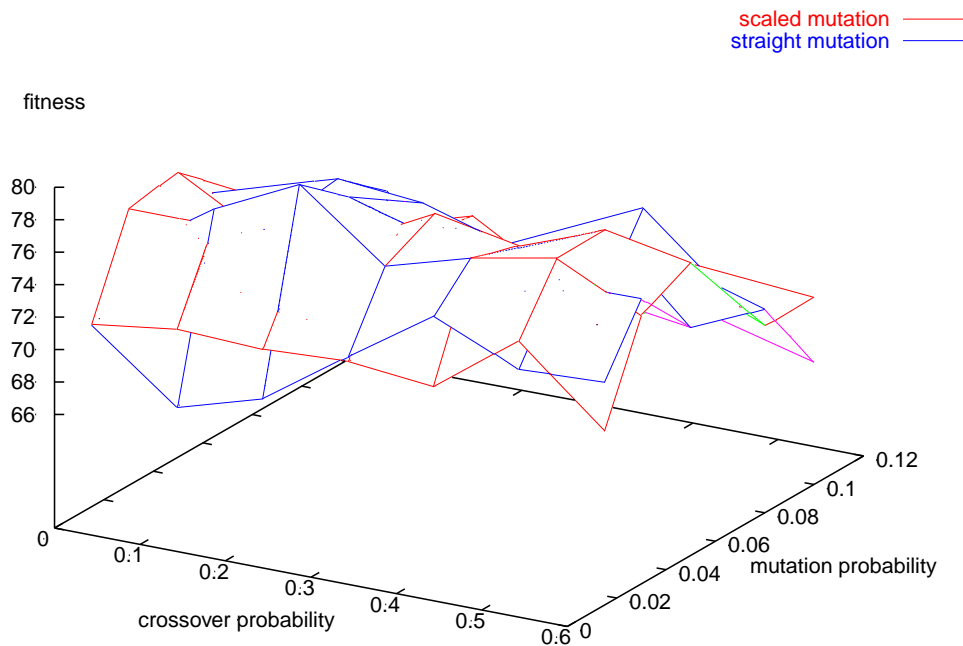


Figure 6.18: Straight versus speed-scaled mutation, test-input16d

scaled are taking turns in being better, however the overall peak is with the straight mutation.

### 6.6.3 Uniform crossover

Uniform crossover shows little overall improvement, as can be observed in figure 6.19 concerning test-input7b. Although test-input9 did show some improvement, the results of test-input16d were worse than with simple crossover.

Most importantly, uniform crossover does not create a larger influence the crossover probability has on the resulting fitness.

## 6.7 Dependency between population size and mutation probability

Let's take a look to see if the population size (which is still inverse proportional to the number of generations since time is kept constant) and the mutation are somehow related.

In figure 6.20 it can be seen that we are indeed looking at a (possibly the) peak of optimal parameter settings for these two parameters. Also it may be noted that the combination of a high population size and a high mutation rate results in a low fitness, while with only one of the parameters

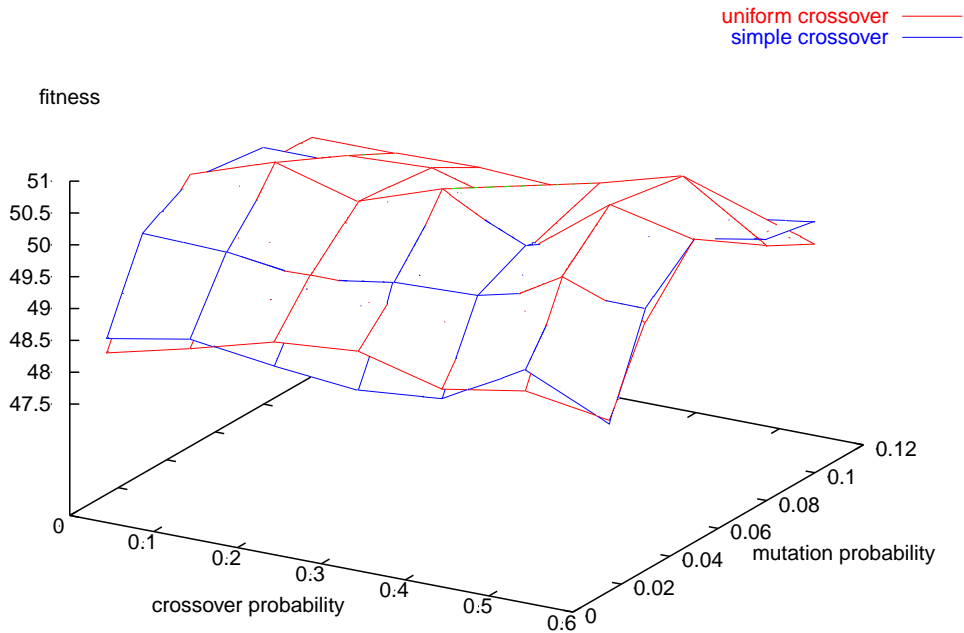


Figure 6.19: Simple versus uniform crossover, test-input7b

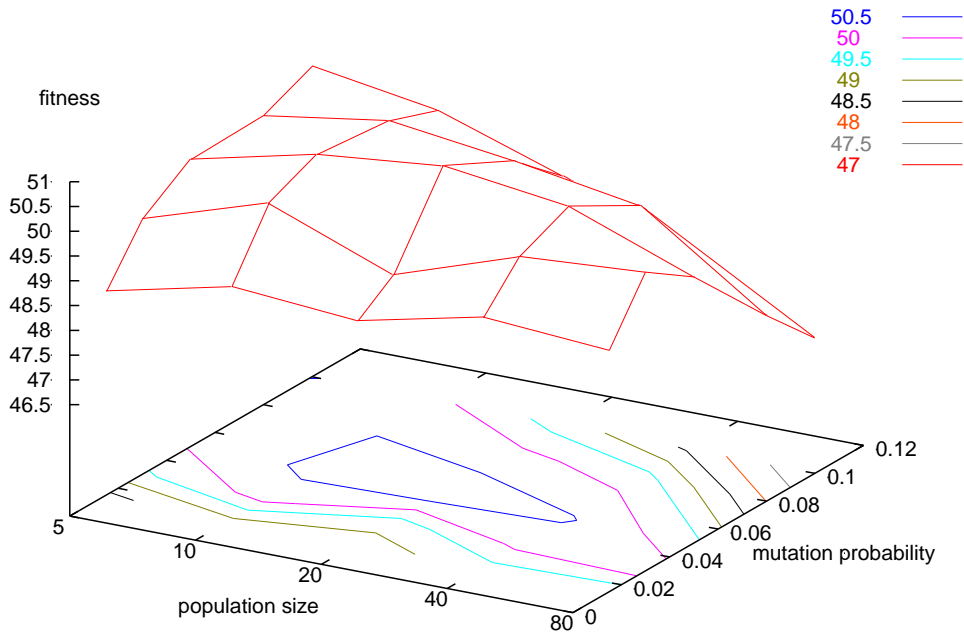


Figure 6.20: Population size/mutation dependency, test-input7b

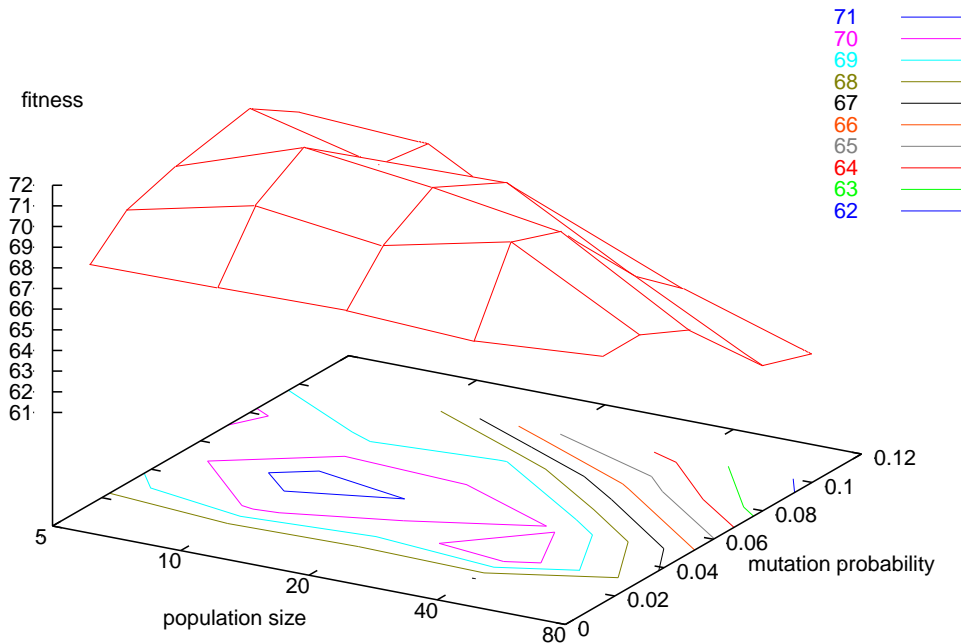


Figure 6.21: Population size/mutation dependency, test-input9

set relatively high, quite reasonable results are obtained. This can easily be seen by the triangle shaped 50.5 contour.

This same kind of triangle shape can be found in the contour lines of figure 6.21, and also a in 6.22, be it a little unclearer because of the more erratic nature of test-input16d.

Overall, a mutation rate of 0.05, a crossover of 0.5 and a population size of 20 seem the optimal settings to gain a good result with all three test-inputs.

## 6.8 Comparing the final genetic algorithm with other scheduling methods

Now that we've come to the optimal parameter settings for the genetic algorithm, it is time to compare the results with the results by non-GA scheduling methods implemented in the performance model, as described in section 5.3.

	GA	GA - $\sigma$	doubleOpt	non-GA	combined	comb. - $\sigma$
7b	50.6365	48.7350	48.2697	49.0774	50.8828	49.0021
9	71.5861	68.1360	69.8191	73.3129	72.6594	69.3041
16d	77.0509	62.0006	64.0000	64.0000	78.0549	62.9027

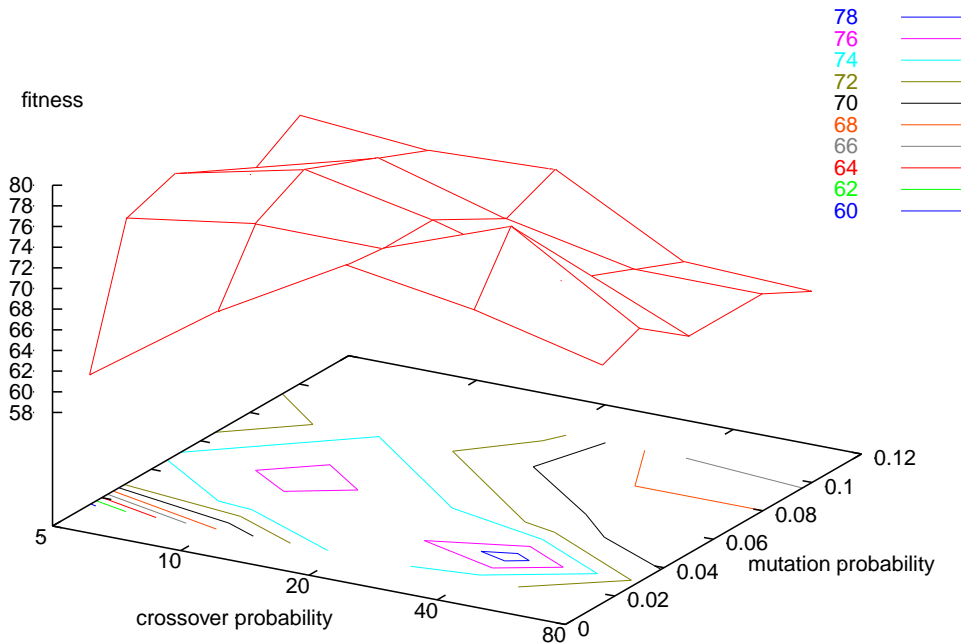


Figure 6.22: Population size/mutation dependency, test-input16d

In the table above, the final results of the genetic algorithm are in the column marked GA. As a reference of how much the result of the genetic algorithm may vary, a column where a single standard deviation is subtracted is shown and has the title GA -  $\sigma$ .

optPerProgramme, lessStupidOpt and doubleOpt are used after each other for the 'non-GA' column. I have also used doubleOpt after the genetic algorithm, and those results are shown in the 'combined' and 'comb. -  $\sigma$ ' columns.

As said before, the genetic algorithm is run for 3 seconds. All other methods were kept running until they were done (without possibility to enhance further), which were all easily under a second.

As you can see, with test-input7b, the differences are small. The genetic algorithm has a very small advantage over the non-GA results.

With test-input9, there is also a small difference, this time the best results are obtained using the combined non-GA solutions.

Finally, with test-input16d, two things are clearly visible: most of the time, the GA generates obviously better schedule than non-GA created schedules. However, the differences between the different runs of the GA are very large, as can be seen by the very much lower GA -  $\sigma$ . GA -  $\sigma$  is even lower than the local maximum (64) that all non-GA algorithms get stuck at, as the genetic algorithm with swap mutation (see figure 6.16) does as well.

As the results of the experiments with test-input16d show, genetic algorithms are not the ultimate answer at solving problems as discussed in section 5.4.3. It definitely did not result in a dual-adjacent-task solution as one might have wanted.



# Chapter 7

## Conclusion

This last chapter summarises the findings of this research, addresses some open issues and gives an indication of what remains to be done.

### 7.1 Research findings

Genetic algorithms are currently not the final answer to the scheduling problem. In 2 of the 3 studied cases here the performance of the genetic algorithm was only marginally different from the alternative approaches.

Also, the non-deterministic nature of genetic algorithms and the large variation of the different runs of the genetic algorithm pose a bad worst-case scenario.

One more issue with genetic algorithms is the time it takes to come to a result compared to other approaches. However, the larger the problems are in the parallel system, the less significant scheduling time becomes relative to the gain of a better schedule. Since computers become faster and faster (without an end in the foreseeable future), the resolution and complexity of the simulation will grow as well. Although I have not tested this, I expect that the genetic algorithm can easily be scaled to this growth and may therefore yield better and better results in time.

Within the genetic algorithm, it proved interesting to see what difference some parameters make. Elitism ensures a big improvement in the selection procedure, and should get more attention in the genetic algorithms field.

Tournament selection proved an elegant and effective method for removing the need of appropriate scaling for the fitness values, since it only compares two values at a time.

The swap mutation gave bad results. It remained stuck at local maxima, as the non-genetic approaches did. This is because of the tendency it has to starve itself of nodes, and has no way to introduce any new ones.

The other alternative genetic algorithm methods I have used did not have a large impact.

## **7.2 Remaining work**

There can still be done much more in this field. I will summarise some of the remaining subjects here.

### **7.2.1 Validation of the model**

First and foremost, the performance model has not been validated in order to be sure it represents the situation in real-life computing clusters correctly.

This is needed to see if the results obtained from this research are applicable in a real cluster of computers.

If not, then after the model is fixed, parts of this research may need to be redone. However, since the genetic algorithm is already available and very much separated from the model, the programming effort on the genetic programming side of things should be minimal.

### **7.2.2 Rescheduling**

All my research time was used for solving the scheduling problem.

For rescheduling, an efficient way will need to be found to favour the genetic algorithm in choosing a situation where not many migrations are made, since the gain of the rescheduling can easily be overshadowed by the cost of the migration itself.

Since the importance of the migration cost over the performance gain is for a large part dependent on how long the situation will remain the same after the migration is passed, some way of looking ahead (for instance, to see if tasks are almost completed) is required.

### **7.2.3 Simulated annealing**

Simulated annealing seems, with its property of making less major changes as time passes, very useful for the rescheduling problem.

This can be applied to the scheduling algorithm currently in use, or in combination with genetic algorithms, making mutation probability shrink while the generation count rises for instance.

## **7.3 Final words**

As can be concluded from the results in chapter 6, genetic algorithms are a viable, but not significantly better, method for creating solutions for parallel scheduling problems if the correct methods are used. This may improve as the parallel programs grow more complex in the future.

# Bibliography

- [1] G.M. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *Proc. AFIPS*, volume 30, pages 483–485, 1967. <http://cs.nju.edu.cn/~gchen/teaching/fpc/amdahl.ps>.
- [2] M. Bubak, W. Funika, D. Żbik, G.D. van Albada, K.A. Iskra, P.M.A. Sloot, R. Wismüller, and K. Sowa-Piekło. Performance measurement, debugging and load balancing for metacomputing. In *ISThmus 2000, Research and Development for the Information Society, Conference Proceedings*, pages 409–418. Instytut Informatyki, Politechnika Poznańska, Poznań, 2000. <http://wwwbode.in.tum.de/~wismuell/pub/poznan.ps.gz>.
- [3] H.M. Deitel and P.J. Deitel. *Java: How to program*. Prentice Hall, third edition, 1999. ISBN 0-13-012507-5.
- [4] L.J. Eshelman, R.A. Caruana, and J.D. Schaffer. Biases in the crossover landscape. In J.D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 10–19. Morgan Kaufmann Publishers, 1989.
- [5] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. PVM 3 user’s guide and reference manual, September 1994. <http://www.netlib.org/pvm3/ug.ps>.
- [6] John L. Gustafson. Reevaluating Amdahl’s law. *Commun. ACM*, 31(5):532–533, 1988. <http://www.scl.ameslab.gov/Publications/Gus/AmdahlsLaw/Amdahls.pdf>.
- [7] K.A. Iskra, Z.W. Hendrikse, G.D. van Albada, B.J. Overeinder, and P.M.A. Sloot. Experiments with migration of PVM tasks. In *ISThmus 2000, Research and Development for the Information Society, Conference Proceedings*, pages 295–304. Instytut Informatyki, Politechnika Poznańska, Poznań, 2000. ISBN 83-913639-0-2. <http://www.science.uva.nl/research/scs/papers/archive/Iskra2000a.pdf>.

- [8] K.A. Iskra, Z.W. Hendrikse, G.D. van Albada, B.J. Overeinder, and P.M.A. Sloot. Performance measurements on Dynamite/DPVM. In J.J. Dongarra, P. Kacsuk, and N. Podhorszki, editors, *Recent Advances in PVM and MPI. 7th European PVM/MPI User's Group Meeting*, volume 1908 of *Lecture Notes in Computer Science*, pages 27–38. Springer-Verlag, September 2000. <http://www.science.uva.nl/research/pscs/papers/archive/Iskra2000d.pdf>.
- [9] K.A. Iskra, Z.W. Hendrikse, G.D. van Albada, B.J. Overeinder, P.M.A. Sloot, and J. Gehring. Experiments with migration of message passing tasks. In R. Buyya and M. Baker, editors, *Grid Computing – GRID 2000: The First IEEE/ACM International Workshop, Bangalore, India*, volume 1971 of *Lecture Notes in Computer Science*, pages 203–213. Springer-Verlag, December 2000. <http://www.science.uva.nl/research/pscs/papers/archive/Iskra2000e.pdf>.
- [10] K.A. Iskra, F. van der Linden, Z.W. Hendrikse, B.J. Overeinder, G.D. van Albada, and P.M.A. Sloot. The implementation of Dynamite – an environment for migrating PVM tasks. *Operating Systems Review*, 34(3):40–55, July 2000. <http://www.science.uva.nl/research/scs/papers/archive/Iskra2000b.pdf>.
- [11] Sami Khuri, Thomas Bäck, and Jörg Heitkötter. The zero/one multiple knapsack problem and genetic algorithms. In E. Deaton, D. Oppenheim, J. Urban, and H. Berghel, editors, *Proc. of the 1994 ACM Symposium of Applied Computation proceedings*, pages 188–193. ACM Press, 1994. <http://citeseer.nj.nec.com/khuri93zeroone.html>.
- [12] Melanie Mitchell. *An introduction to genetic algorithms*. MIT Press, 1996. ISBN 0-262-13316-4.
- [13] Stuart J. Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Prentice-Hall, Inc., 1995. ISBN 0-13-360124-2.
- [14] G. Syswerda. Uniform crossover in genetic algorithms. In J.D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 2–9. Morgan Kaufmann Publishers, 1989.
- [15] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, Inc., third edition, 1996.
- [16] S. White, A. Ålund, and V. S. Sunderam. Performance of the NAS parallel benchmarks on PVM-based networks. *Journal of Parallel and Distributed Computing*, 26(1):61–71, 1995. <http://www.nas.nasa.gov/Research/Reports/Techreports/1994/PS/RNR-94-008.ps>.