TU-Berlin
Institute for Communication and
Operating Systems

Diploma Thesis

Dynamic Load Balancing in Heterogenous Cluster Systems

Philipp Fechteler <ruffbg@cs.tu-berlin.de>

January 12, 2005

Supervised by: Barry Linnert
Prof. Dr. Hans-Ulrich Heiß

Die selbständige und eingenhändige Anfertigung versichere ich an Eides statt.
Berlin, den

Unterschrift

Contents

1 Introduction

Ever increasing demands for computing power and, at the same time, decreasing costs for simple computers[1] have evolved cluster computer systems. Cluster computer systems consisting of interconnected computers coupling their compute power and have become very popular and easy to set up. Though, the advantage of more computing power also introduces more complexity in software development for such machines.

In order to *speed-up* the execution time or to *scale-up* the problem to solve, the program has to be distributed on several processors. Prior to the distributed execution, the parts of the software that can run in parallel have to be identified and mapped onto processors. Currently, the software developer himself has to divide his program into several processes in order to let them run in parallel. Moreover, the operating system has to efficiently map the processes to processors, which are part of the cluster system.

Today's systems supporting the execution of parallel programs such as *MPI*, see [MPI], assign the whole problem to the developer. The developer has to explicitly hard code the mapping of processes to processors when writing software. Due to the complexity of this task, the mapping is mostly very simple and highly optimised for one special hardware environment. In the majority of the cases, the mapping requires the exclusive usage of the whole hardware, respectively of a partition assigned to the program to be mapped. This leads to very static and inflexible solutions that do not support new hardware environments.

A better approach to develop parallel programs would be to split the process into several sub tasks. The program developer then just develops the software with exploitation of any parallelism. He therefore specifies all parts that can run in parallel by using parallel statements and ignores all aspects concerning any target machine. A later task is to identify all the parallelism stated within the parallel pieces of the program and to generate concrete processes out of them. The subsequent task is to map these processes to processors. The mapping is done at program start. Therefore, the current state

[1] simple computers compared to highly optimised parallel supercomputers, like Crays

and structure of the cluster computer can be taken into account. Doing the mapping not until the program start removes the binding to one particular homogeneous cluster computer. Furthermore, this solution avoids the need of exclusive, static partitions. The proposed solution additionally allows dynamic load balancing, which optimises the utilisation of the cluster system both, in time and space.

The goal of the framework introduced in this diploma thesis, as well as [Sch04] and [Gra04], is to overcome the problems described. The framework will automate the whole process of developing parallel programs in the way such that the developer just writes software containing the desired algorithms. The framework will identify the parallelism within the program. The mapping unit of the framework will map all parallel tasks onto processors at program start. It will take into account the current state and properties of the hardware[2], the dependencies and the communication structure of the program. The framework realises dynamic load balancing. In order to do so, the loads of the processors will constantly be watched so that the mapping unit can directly react on load changes of the processors.

This thesis is concerned with the latter mentioned dynamic load balancing and mapping of processes to processors being part of a heterogeneous cluster system. The other tasks are already discussed: [Gra04] deals with the on- and offline detection of parallelism in programs and [Sch04] discusses the generation of a model out of measured values for the mapping unit.

This document is structured in the following way: The environment of the topic of this thesis is described in chapter 2. In chapter 3, some terms, models and assumptions are introduced, respectively clarified. Subsequently, the new approach taken by this work is presented in chapter 4. In chapter 5, practical issues and evaluations are elaborated on. Finally, in chapter 6 are further aspects presented which arose during the work and should be examined in the future.

[2]load and power of the processors, bandwidths and latencies of the communication network etc.

2 Parallelising of Programs

Distributed cluster computers are used to *speed-up* the execution time or to *scale-up* the problem which is to be solved [Hei94]. Therefore, cluster computers consist of multiple computer nodes usually having multiple processors. All of these processors can be used to simultaneously work on different parts of a computation problem.

In order to benefit from having multiple processors within a cluster computer, the software has to be adapted so that it utilises more than one processor. Calculations, which can run in parallel have to be simultaneously executed on different processors. Therefore the calculations will be divided into processes, which will be executed on the processors of the concrete cluster computer, the target system. Cluster computers are often operated in *space-sharing* mode in order to permit several users access to its resources. The space-sharing is done by dividing the cluster into so called *partition*s. A partition can then be assigned to the user, who can exclusively run his programs on it.

2.1 Conventionally Proceeding

There are generally three steps necessary to put an algorithm into execution on a concrete cluster computer [RR04], see figure 2.1:

- division
- agglomeration
- mapping

The first step is to divide the amount of calculations into smaller pieces (*tasks*). The amount of resources and the capabilities of the cluster computer have to be taken into account during the process of such a division. This means that the developer must not divide the algorithm into more pieces

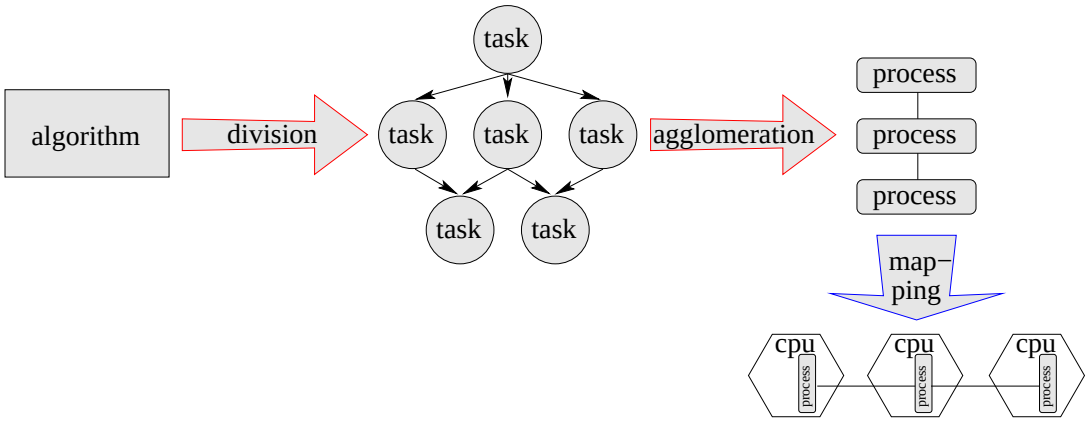


Figure 2.1: common tasks involved when developing parallel applications; the red arrows indicate the programmer's task, the blue one describes the work of the computer system

than the actual number of processors available on the cluster computer, respectively the partition. The originated tasks can depend on each other, e.g, when a task depends on the result of another task. After the first step, the splitting up of one sequential computation into parallel computations, the programmer should have a very good understanding of the tasks and their inter-dependencies.

In the next phase, right before the implementation, the programmer builds processes out of the tasks. This is called agglomeration [Fos95]. Currently, it is conventional to do this under consideration of the concrete target machine on which the program will be executed [RR04]. The goal of this agglomeration is an optimal distribution of the computational load to the processes and the communication between them. The load of computation is seen as sum of all possible calculations of each process and should be uniformly distributed relative to the capacities of the processors of the target cluster system. This shall guarantee that all processors are busy and no process delays the computation time of the whole program because of a bold execution time. Another criterion is the behaviour in terms of communication. Processes should not exchange too much data because they could be executed on different nodes. In that case, the data would have to be transmitted time-consuming across a network. This issue can be avoided by assembling together the tasks which work on the same data into one process to ensure the locality of data.

Furthermore, we can distinguish between static and dynamic agglomeration. Static agglomeration means that the division is the same every time, whereas dynamic agglomeration means that it can be changed by the parallel program during runtime. Though, in case of dynamic agglomeration, the main decisions are still made by the programmer.

The last step is the assignment of the processes to the processors of the target cluster computer. This is called the mapping. Thereby, the operating system tries to find an optimal usage rate of the processors. If the programmer, as it is common nowadays, has created just as much processes as the computer possesses processors, the mapping is trivial.

This is the main advantage of the current method. The operating system does not need any further information about the program in order to make such simple mapping decisions. As previously stated, the optimal distribution of communication and computational load is the matter of the programmer. So on the one hand, he has full control over his parallel software, but on the other hand, this introduces additional effort for the programmer. Another drawback is that the program is optimised for one particular cluster system. If the program will later get a smaller or bigger partition of the computer or if it actually gets executed on a completely different machine, the operating system has difficulties to find an optimal mapping.

2.2 New Framework

The goal is now to take off the responsibility for an optimal mapping away from the programmer. At the same time, the operating system should be able to carry out the mapping as optimal as possible. The difference to the previous method gets clearer by reviewing the previously mentioned three steps.

The programmer still has to divide his software into tasks. Though, now it should be done accurately enough for the problem, without the consideration of any, now generic, target machine.

The second step is again the partitioning of the tasks into processes. But now, it is desirable to get a high degree of parallelism meaning that there should be as many tasks in separate processes as possible. This is not very complex with the preceding modelling of tasks and dependencies. Hence, the programmer can concentrate more on the actual program.

With such a program, there are no problems running it on different architectures. In most cases, the operating system has to assign more processes than processors available. Hence, it can influence the distribution of the load. It is also conceivable that an assigned partition changes at mapping

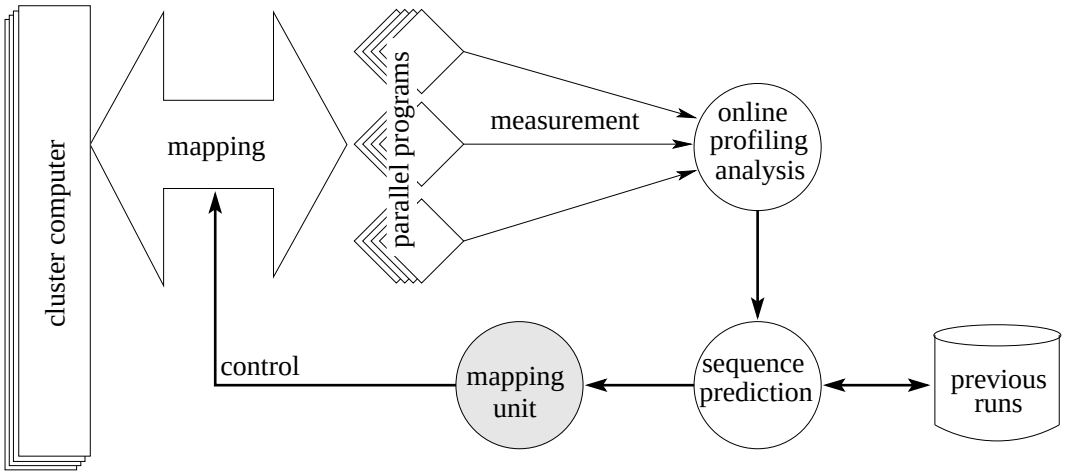


Figure 2.2: new framework with an automatic mapping of tasks of parallel programs to processors of a cluster computer

time or even dynamically during runtime. Such a change requires and triggers a remapping of the processes to the new set of available processors. The system described above realises dynamic partitioning [FRS97], which means that the partitioning depends on the real load of the processors. Due to this dynamic partitioning the operating system ensures an optimal mapping of tasks to the concrete machine.

Though one drawback of this proceeding is, that potential knowledge of the programmer, which would have been used for the mapping in the previous proceeding will be unconsidered now. E.g. the programmer has to provide locality of data but the operating system cannot know which tasks in which processes access the same data. And likewise, the programmer can give better estimates for the runtime behaviour of the tasks. Furthermore, additional costs arise for the parallel execution of tasks on one processor in contrast to the sequential execution because of context switches for instance. This can be avoided running the processes one after another, likewise Gang-Scheduling. But therefore, the operating system would need additional information about the processes and their behaviour.

The approach is now to increase the flexibility of the operating systems mapping without having the disadvantages mentioned above. Therefore the system needs additional data about the runtime behaviour of the program, which the programmer described more or less well. In the new framework, there should also be automated predictions made about the program's behaviour. Such knowledge would help the operating system to make the map-

ping. The information needed can be supplied by the programmer or can be extracted out of the program. Because the programmer should be relieved, the latter approach will be examined.

The objective is that the operating system gets an automatically created prediction of the presumable behaviour of the program. This prediction should serve as the basis for an appropriately extended mapping unit and should originate from measurements of previous runs.

The implementation of this approach, shown in figure 2.2, can be divided into three parts:

measurement measured data of single program runs must be prepared to represent the computation and communication load as well as the structure of the sequence

prediction based on this data a model of one concrete sequence must be build, and out of a set of such models a prediction for future runs must be created

mapping the tasks of a program will be mapped and dynamically remapped onto the nodes of a concrete cluster system under consideration of the prediction for it and the state of the machine

Methods for suitable measurements and a presentation of their results can be found in [Gra04]. Possibilities for the prediction based on graph transformations are discussed in [Sch04]. Mapping and load balancing is the topic of this diploma thesis.

3 Basics and Theory

For the development of the mapping unit and the load balancing the demands on such a system should be discussed. Hence a measure of quality have to be defined as well as a method to determine it. This presents the aims of this chapter in addition to the clarification of some ambiguously used terms and the discussion of the underlying basic models.

3.1 Technical Terms and Notations

Terms like task, thread and process are often used having different meanings. As this work supplements the framework introduced in [Gra04] and [Sch04], the opportune terminology used in there will be adapted here. The resulting definition of these terms, illustrated in figure 3.1, is:

task amount of instructions and function calls with a common semantic context

process amount of tasks which run in sequence

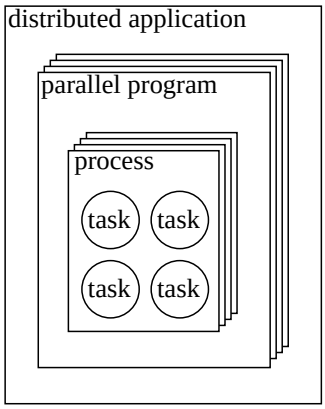


Figure 3.1: classification of terms

parallel program amount of processes which run in parallel or sequence

distributed application amount of parallel programs which corporate to deal with a common job

If programs are mentioned in this thesis parallel programs are meant. This work is concerned with tasks of parallel programs and their relations. Furthermore, tasks are the elementary entities of mapping and load balancing.

For the mathematic formulas in this work, a notation in the style of object oriented programming is used. Hence, there are objects with attributes and methods. To reference the methods or variables of an object o the following syntax is used:

o.**var** usage of variable "var" of object instance o

o.**proc()** application of method "proc()" of object instance o

In this work there will be mainly task and processor objects. They will be introduced in chapter 3.3, after the discussion concerning the modeling aspects.

3.2 Modeling of hard- and software

This section describes the abstractions which were done for the used algorithms, the models and the underlying data formats.

The model of a cluster computer is shown in figure 3.2. It is represented by a graph called *machine graph* or *hardware graph*. A cluster computer consists of several connected processors, which are represented by nodes in a machine graph. The processors are characterized only by their clock speed or rate, like 3 GHz. A network between the processors defines the communication connections. Each connection is unidirectional and specified by bandwidth and latency. The communication connections are mapped to directed edges in the machine graph. Processors of one node which share there memory and have therefore a very fast connection have no special notation in this model. They will be represented simply by an edge with a very high bandwidth and a minimal latency.

Hence a whole cluster computer corresponds to one hardware graph. All of its processors correspond to nodes in the graph in a 1:1 relation. The nodes of a cluster systems, which may contain several processors, are not directly contained in the machine graph. They can be identified by edges

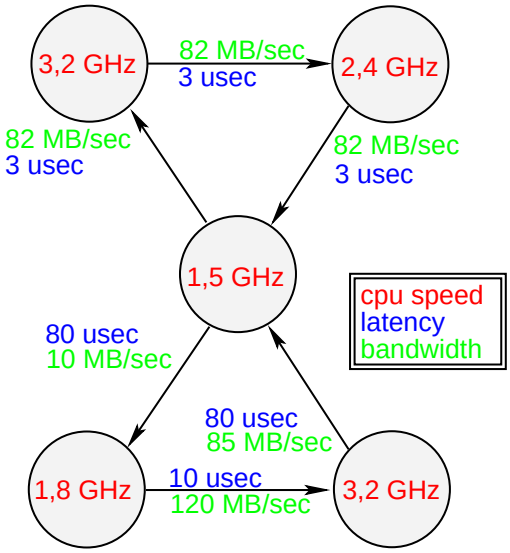


Figure 3.2: hardware or machine graph

with a very high bandwidth respective a very low latency. The remaining edges represent the communication network of the cluster system.

There is no information about cache or memory contained in this model. Though, it would not be too complicated to retrofit it. Cache and memory were omitted because they would increase the problem of mapping and load balancing. Its absence does not disturb the study, because the main focus of this work is on the execution and communication behavior of parallel programs. Including them could be a later task to do.

This model scheme allows heterogeneous cluster computers to be easily modeled.

The model of parallel programs to be run on a machine graph are also represented by graphs, shown in figure 3.3. These graphs are called *software graphs* or *program graphs*. The nodes of a program graph comply with the tasks of the program and have the amount of instructions as attribute. There are two types of directed edges between nodes. One represents communication and the other one represents a dependence. Both edge types have an amount of bytes as label. The communication edge represents the data, which the source of communication sends to the receiving task that has to run in parallel. The dependence edge represents the data, which the originating task leaves behind for the receiving task. The receiving task cannot start until it has received all of the data it depends on.

Based on these two models, the task of mapping is to assign every task of every program to a processor of a machine. The criteria to measure the

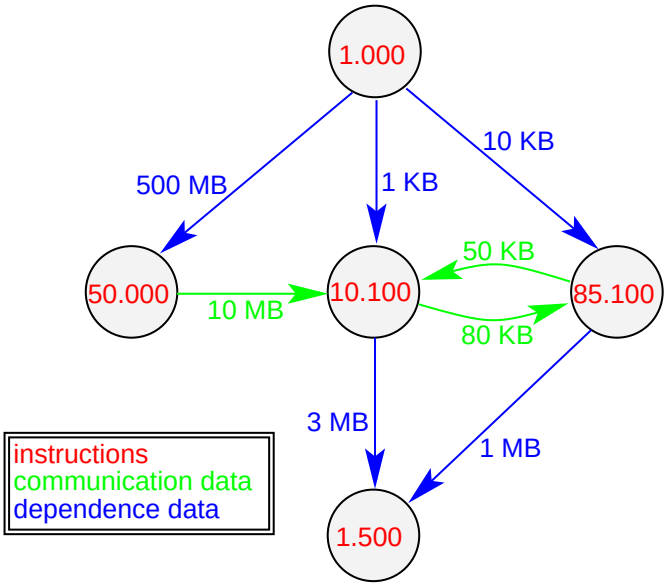


Figure 3.3: software or program graph

quality of the mapping is discussed in detail in chapter 3.5. The runtime of any task is deterministically determined on the one hand by the amount of instructions and the corresponding processor power, and on the other hand by its size of communication data and the bandwidths and latencies of the corresponding communication paths.

This is one main difference to the model of the load used in [Sch91] on which this work bases partly. The model used there consists of phases. Each phase consists of a constant amount of processes, which maintained deterministic communication relations. But the transition of the system from one phase to another is stochastic in both senses: the moment of transition and the choice of target phase.

3.3 Syntactical models

For the mathematic formulas in this work an object oriented syntax is used, which was introduced above in chapter 3.1. It will be used to describe objects out of the presented models above and their behavior. There are task objects with attributes for instructions, communications and dependences. There are also processor objects with attributes for their power, the number of tasks assigned to that processor and the amount of programs the tasks belong to.

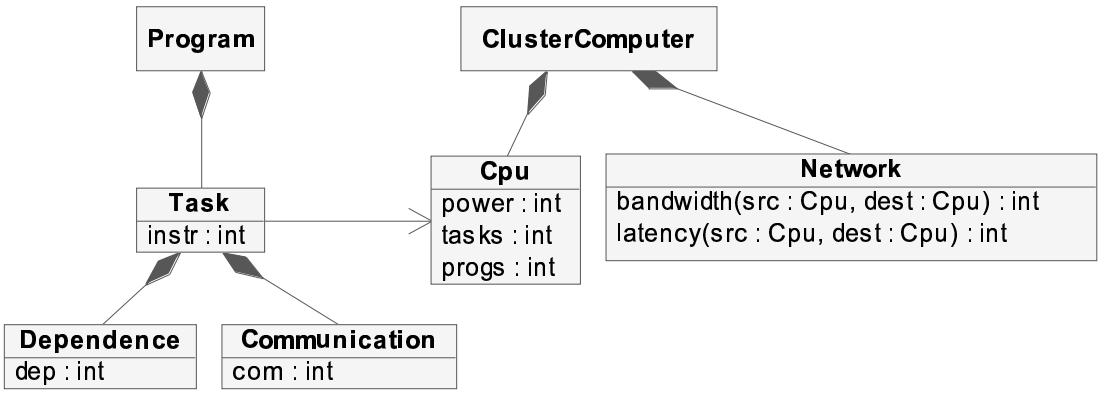


Figure 3.4: idea behind the used notation as UML diagram

The notation, illustrated as UML[1] diagram in figure 3.4, is as follows:

t.**instr** amount of instructions of task t

t.**cpu** processor task t is assigned to

t.**com**$[c]$ amount of dependence data task t will send to task c

t.**dep**$[d]$ amount of dependence data task t will send to task d

c.**power** power of processor c

c.**tasks** amount of tasks assigned to processor c

c.**progs** amount of programs which have tasks assigned to processor c

bandwidth$(c_{from} \rightarrow c_{to})$ bandwidth of the connection from processor c_{from} to c_{to}

latency$(c_{from} \rightarrow c_{to})$ latency of the connection from processor c_{from} to c_{to}

The task objects always belong to a program and the processor objects belong to a cluster computer. The procedures bandwidth() and latency(), which are used without a referencing object are implicitly part of a cluster computer object.

[1]UML - Unified Modeling Language

3.4 Simulator

A simulation of a cluster system is needed because the mapping and load balancing problem is to complex to overcome it by using analytical methods. In addition, a cluster computer for extensively testing is not available. Hence, for verification of the approach presented in this thesis, a simulation of a generic cluster computer was implemented. This simulation enables the user to load a description of a machine graph in order to instantiate a simulation object. Several descriptions of program graphs can be loaded into this simulator object. The user is able to step through the execution of such a loaded program graph. Furthermore, any implemented mapping and load balancing algorithm works with such an object.

The simulation of cluster systems does not contain the control flow of the parallel program itself but the action taking place on the processors and the communication network. This means that architectural issues of the algorithms, i.e. usage of a centralized or decentralized solution is beyond the scope of this work. Therefore mapping and load balancing costs no time in simulations. This is, likewise the lack of modeling memory and caches in the used models, only a small drawback. Because this study shall only demonstrate that there may be advantages by using the framework, the simulation of the control flow was omitted for simplicity of the simulator. This work is not meant to be a quantitative analysis. This could be done in further work.

The instruction count of running tasks assigned to a processor decrease homogeneous over simulation time. This means that all running tasks of a processor get equally sized and very small slices of processor time slices without any scheduling costs. An alternative view would be that a processor splits its power into equal sized slices, for every running task, and lets every task run with such a slice of processor power.

The simulation calculates one clock cycle for every instruction. Hence, the runtime of a task is determined by the amount of instructions. Though, there is one exception, in case when the communication of a task lasts longer than the runtime of instructions. In this case, the task runs as long as it communicates with other tasks and consumes processor power. The correspondence in the alternative view is, that the task still consumes its slice of processor power. This is because of the fact that a communicating task executes instructions.

Every time, a simulation object is instantiated, a description of a machine graph is passed to it and a table is constructed out of this graph. The table contains the bandwidth and the latency between each pair of two processors.

For that, the path with the highest bandwidth between processors is searched and the corresponding latency of that path is taken. The bandwidth of each path is determined by the bandwidth of the smallest connection between two neighboring processors in the path. If the processors of every path p of the set of all possible paths P between two processors c_{from} and c_{to} are individual numbered, then the formula for the bandwidth between them is:

$$\begin{aligned}
\text{bandwidth}(c_{from} \rightarrow c_{to}) &= \max_{p \in P} \text{bandwidth}(p.\text{cpu}_{\text{from}} \rightarrow p.\text{cpu}_{\text{to}}) \\
&= \max_{p \in P} \left[\min_{i=\text{from}\ldots\text{to}-1} \text{bandwidth}(p.\text{cpu}_i \rightarrow p.\text{cpu}_{i+1}) \right] \\
p^{best} &= \operatorname*{argmax}_{p \in P} \text{bandwidth}(p.\text{cpu}_{\text{from}} \rightarrow p.\text{cpu}_{\text{to}})
\end{aligned}$$

The latency of the path p^{best} with the best bandwidth is simply the sum of all connections:

$$\text{latency}(c_{from} \rightarrow c_{to}) = \sum_{i=\text{from}}^{\text{to}-1} \text{latency}(p^{best}.\text{cpu}_i \rightarrow p^{best}.\text{cpu}_{i+1})$$

The simulator abstracts from the real usage of the communication network. It assumes that path p^{best} between two processors is always free for exclusive usage. Thus, the parallel usage of multiple paths for one communication[2] is not taken into account. Likewise the parallel concurrent usage of one connection by multiple tasks for several communications[3] between processors is not considered. This is done as a first simple approximation of the real communications. The topic of finding a route through a network and balancing of communicational load represents an own field of research.

3.5 Design Criteria and Decisions: Goodness & Performance

The mapping and dynamic load balancing have to be done because the "simple" assignment of a static partition is not sufficient in most cases. Figure 3.5 shows a possible profile of the processor usage of a program. The figure illustrates how heterogeneous the degree of parallelism can be and how unsuitable such a program behavior is for one static partition.

[2] communication from one task to one another task over multiple paths

[3] communication of several tasks to several other tasks over the same connection

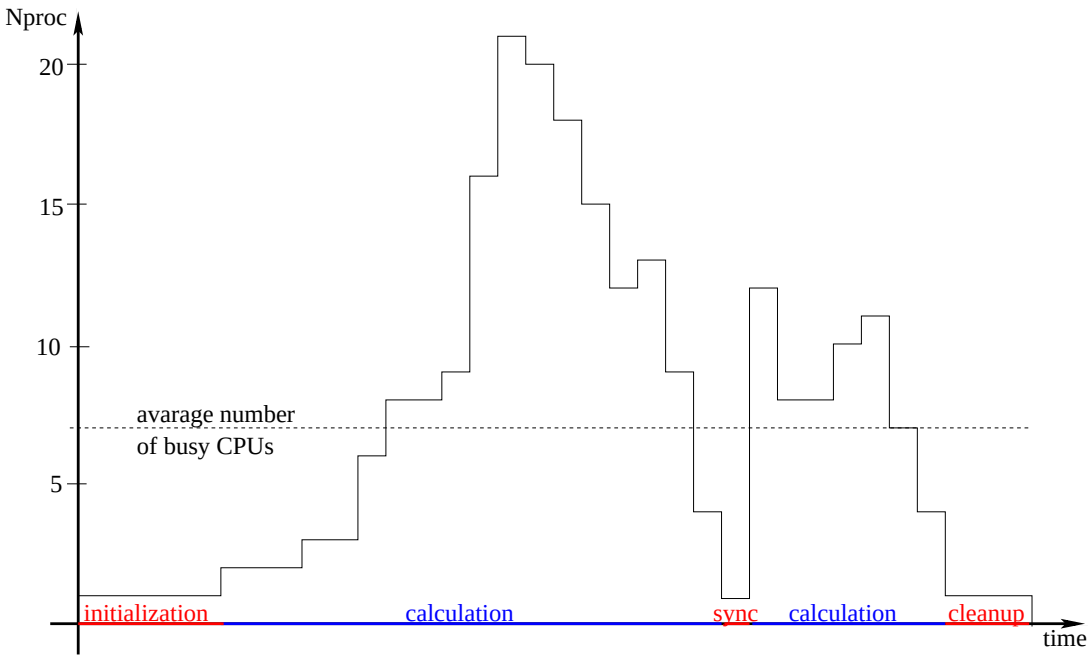


Figure 3.5: possible profile of parallelism for a program

Of course, it is desirable that mapping and load balancing are done as optimal as possible. But there is no commonly used measure of quality for the given problem. To some extent, there are oppositional demands on such a measure, depending on the view [Hei94]. The user, who starts his application on a cluster computer possibly wants minimal response or execution times as well as a maximal speed-up, respectively scale-up. The operator of such a machine wants to maximize the throughput and utilization and minimize the load unbalance.

One compromise is to build a weighted sum of measures. With this approach it would be possible to satisfy several demands with different weights.

Another compromise is the following function, called *Beschleunigungseffizienz* [Hei94]. It is the product of the speed-up and the efficiency, which is itself the speed-up normalized to the amount of processors:

$$\eta(p) = E(p) \cdot S(p) = S(p) \cdot \frac{S(p)}{n} = \frac{S(p)^2}{n}, \text{with } S(n) = \frac{T(1)}{T(n)}$$

In this formula $T(i)$ denotes the execution time of a given program running on i processors. The shape of this functions for an example problem is shown in figure 3.6. A good mapping according to the Beschleunigungseffizienz results in its maximum.

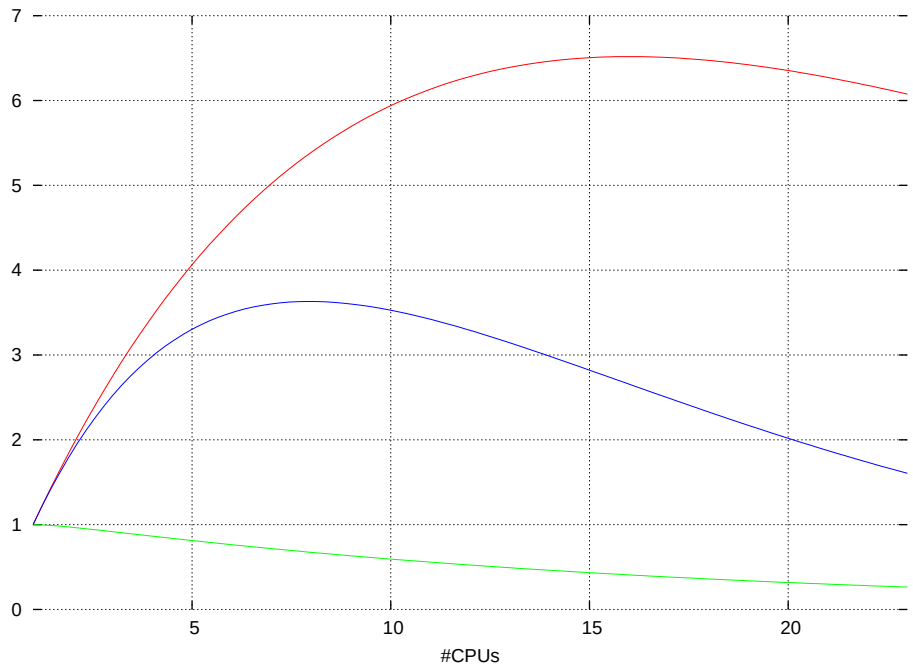


Figure 3.6: relations between speed-up (red), efficiency (green) and Beschleunigungseffizienz (blue)

A drawback of the mentioned quality measures is that they only take the amount of processors per program into account. Both, information about communication and possibly heterogeneous hardware remains unconsidered. Additionally, all these measures are quite theoretical and in practice often difficult to measure. This is why, in this thesis, only the simple mean response time is measured for every program which was running. The mean response time is normalized by the program's number of tasks. The smaller this value is for a given problem, the better the mapping was. As formula:

$$\text{meanTaskRunTime} = \sum_{i=1..N} \frac{\text{Prog}[i].\text{runTime}}{\text{Prog}[i].\text{tasks}}$$

The idea behind the normalization to the number of tasks is that the tasks should be small entities which do not vary to much in size. Size here stands for costs for executing instructions, communication and transmission of dependence data.

Another criterion, which must be considered is the time the mapping, respectively the load balancing needs itself. A potential algorithm should be efficient. This means that it should be at least of a smaller complexity than NP, which is the brute force approach to get the optimal solution:

simulate every combination of mapping and choose the optimal result. Such an approach would take $\#\text{processors}^{\#\text{tasks}}$ tests.

A consideration is as follows: permit the mapping to take more time, because it runs just one time at program start and enforce low timings for the decisions of load balancing which will run possibly very often. In practice, the amount of time the mapping phase takes should not be to large in comparison to the runtime of the program.

4 Combination of Static and Dynamic Approach

The following chapter presents the combined approach of a mapping unit and the mechanisms of load balancing. The mapping unit provides a static mapping prior to the start of the program, whereas the dynamic load balancing remaps tasks respective migrates them during the runtime of the program. The latter mechanism is triggered by changes in the load of the cluster computer. It shall ensure the balancing of load, as the name already says.

The following sections start with the discussion of some simple algorithms which the combined approach will compared with. Subsequently, the combination of the two methods will be described. This will be followed by an introduction of the algorithms used in the particular methods.

4.1 Conventionally & Optimal Scheduling Algorithms as References

In order to have some algorithms to compare the results of the static and dynamic combination algorithm with, two variants of a very simple mapping method as well as the algorithm for the optimal solution were implemented.

A simple method to realise the mapping is to assign every starting task to the processor with the highest clock rate to load ratio. An improvement of this algorithm is to divide the clock rate by the amount of tasks assigned to this processor. For a cluster computer with a set of processors C this can be formulated as[1]:

$$\max_{c \in C} \frac{c.\text{cpu.power}}{c.\text{cpu.tasks}}$$

The complexity of both algorithms is, depending on the implementation, $O(\#\text{processors})$. The difference between the two variants is, that the former method just takes the currently running tasks into account. The latter one

[1] the definition for the notation can be found in chapter 3.1 respectively 3.3

also considers tasks, which are already assigned to a processor but are not yet running. Such tasks are waiting for dependence data and will probably soon be running and consume processor power. The disadvantages of both variants is that the network usage is not taken into account. This means that they do not incorporate the structure of the programs concerning dependences and communications. Another drawback is the fact that it remains unregarded if some running tasks will finish soon so that the corresponding processor will soon have more processor power to provide. Hence, these algorithms serve just as a broad clue for the behaviour of a parallel program.

The algorithm for the optimal solution is simply to run the simulation with every combination of task mapping and picking the best one. Clearly, the massive drawback is that this approach is NP-complete. This means that for the problem of mapping n tasks to m processors there are m^n possible solutions, which have to be calculated to find the optimum. For illustration of the problem: the mapping of 15 tasks to 8 processors, which are both very small numbers, results in $8^{15} = 35,184,372,088,832$ potentially solutions. Figure 4.1 shows the measured runtimes[2] of the algorithm for different problems. The runtime ranges over $18\frac{1}{2}$ hours for 8 tasks on 11 processors. So running much bigger problems will be impossible.

4.2 Combination of Static and Dynamic Approach

The approach of this work is composed of two phases: a static global phase for mapping the tasks to processors prior the program start, and a dynamic local phase for load balancing during the runtime. The phases are combined because of the demand for a method, which can manage the resources of a cluster computer dynamically for several users simultaneously. As neither the pure static mapping, nor the dynamic load balancing itself performs very well in this case, the advantages of both methods shall be combined in this approach.

The static phase is illustrated in figure 4.2 and is at program start before any task has started. In this phase, every task of the program will be assigned to a processor. The static approach bases on the load situation of the cluster computer at the moment and the structure of the program, the dependences and communication relations. Its aim is a globally optimal solution. But, because of the discussed complexity, only heuristic methods can

[2]measured under Linux with the system call getrusage() on an Athlon 1800 XP processor

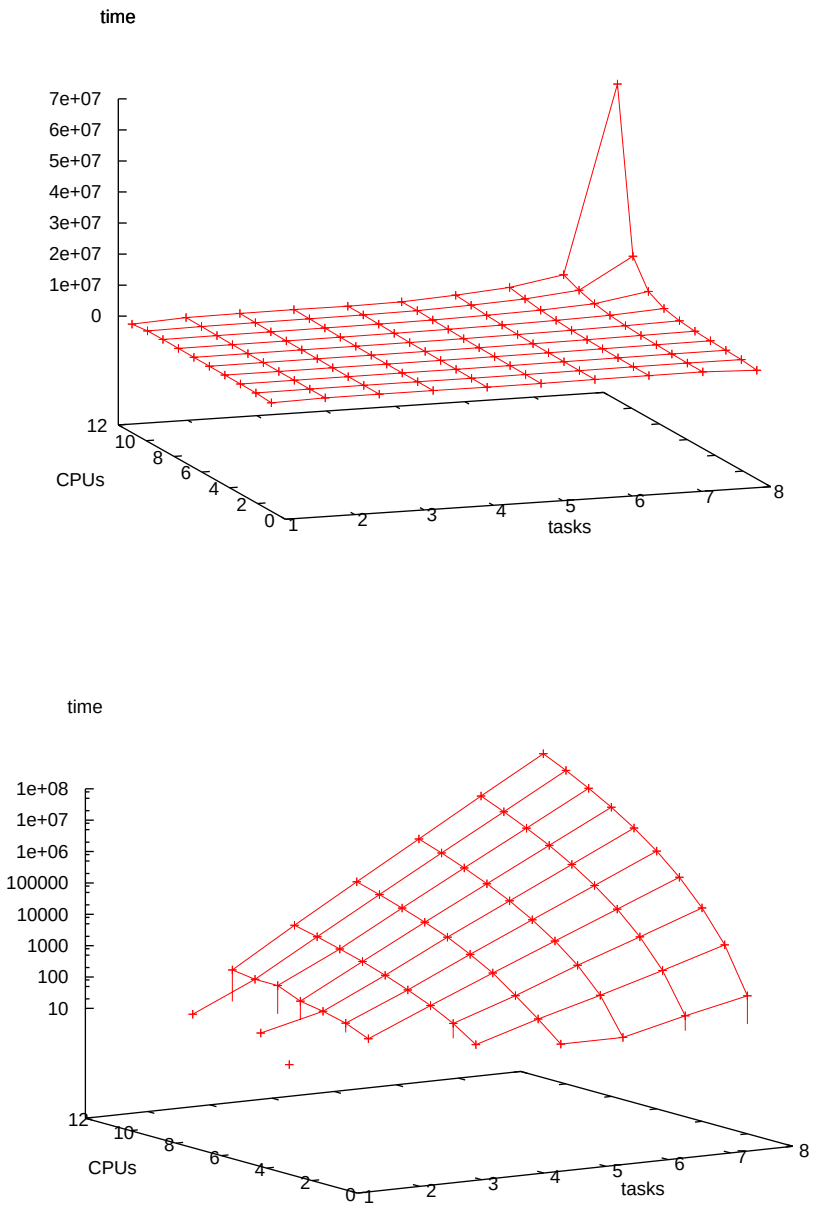


Figure 4.1: measured runtimes in msec for the generation of the optimal solution for different sized problems, above: graph with linear time scale, below: logarithmic time scale

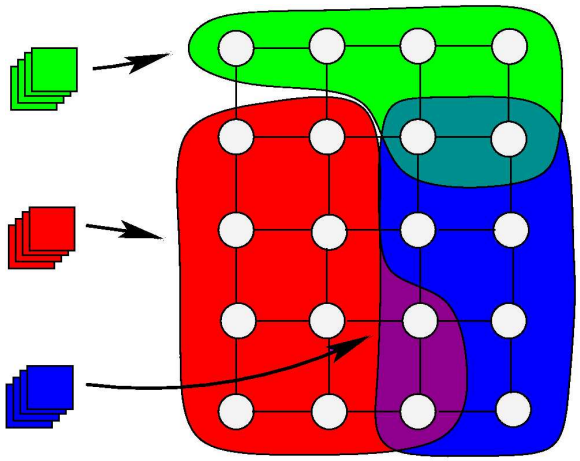


Figure 4.2: static phase of the combined approach

be used which will find nearly optimal solutions. An assumption which is integrated in this mapping, is a constant load situation of the machine during the runtime of the program. Hence it represents a static mapping which is meant to be a broad assignment. Fine tuning will be done in the dynamic phase.

The dynamic phase, shown in figure 4.3, will be performed at the whole runtime. The dynamic approach is responsible for the handling of any dynamics of all programs and the state of the hardware. Therefore, at start of every task, except the initially started tasks, the two assumptions of the static phase will be proofed for validity. They are:

- load of every processor does not change the whole runtime of the program
- program behaves according to the software graph

The consequence is that a readjustment has to be done if the processor, a task was assigned to, has not the same load that it had during the static phase. Another case in which a remapping has to be considered is when a task has more or less children than specified in the corresponding software graph. If these assumptions still hold on the task will start on the processor it was assigned to. If they do not hold any longer, then the mapping will be readjusted locally. It will be checked if the task should start on the processor it was assigned to during the static phase or if it should start on a neighbouring processor.

The remapping of a task from one processor to another one represents the load balancing mechanism of this framework. The transfer of the dependence

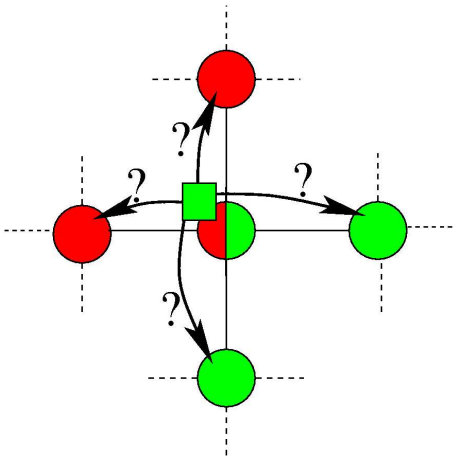


Figure 4.3: dynamic phase of the combined approach

data corresponds to the migration of a process in conventional cluster systems which realise load balancing. The smaller the tasks are, the better is the approximation of this mechanism to dynamic load balancing.

4.3 Static Mapping

In the static phase every task of a just starting program will be assigned to a processor of a cluster computer. The mapping is done with respect to the structure of the program and the current state of the cluster computer, which is supposed to be constant during the runtime. This assignment is done by mapping a software graph into a hardware graph, whereas the software graph is the constant description of the program behaviour. The hardware graph consists of the graph corresponding to hardwares structure, though the values are adapted to the current state of load.

The mapping should be done as optimal as possible. In the given case, it is subjected to the minimisation of the mean response time of the tasks. This leads to a NP-complete combinatorial optimisation problem, as the Travelling Salesman Problem (TSP)[3]. Hence, all optimal algorithms which solve this problem will have an exponential complexity[4]. Because the problem has to be solved at every start of any program, it should not consume to much time as well as resources of the cluster computer. This results in the usage of heuristic algorithms. They tend to be much faster at the price to find

[3]Travelling Salesman Problem - a salesman who wants to know a course across several towns by minimal travailing costs, meeting each town only once

[4]assuming P = NP

only near optimal solutions. These algorithms search in different manners the minimum of a given *cost function*. The cost function is very critical in this process and have to be defined in order to get a solution. There are a lot of algorithms for such tasks. The following is only a small cut out of the diversity:

- Gradient Descent
- Genetic Algorithm
- Hopfield Neural Networks
- Simulated Annealing

The following sections will present the some possible variations of the cost function for the given task. Subsequently, the algorithms used for the static mapping will be discussed. Finally, the last section shows the execution behaviour, if the mentioned assumptions of the static mapping are broken.

4.3.1 Cost Function

Like all optimisation algorithms, Hopfield Neural Networks and Simulated Annealing both minimise a cost function, which must be defined. The cost function is often also called energy function in analogy to physical systems since physical systems tend to stabilise in states of minimal energy. In our case, the cost function is a sum of weighted terms, each of which is normalised to the range of $[0 \ldots 1]$. Each term stands for a different partial goal described in the following:

minimise execution time: fast processors are preferred

promote distribution onto many processors the fewer tasks of the same program are running on the same processor, the more preferable is the mapping

minimise costs of communication the less the sum of communication time between the tasks on the processors of the mapping, the preferable the mapping

minimise costs of dependences analog to the communication costs: the less the sum of transmission of dependence data between the processors of the mapping, the preferable the mapping

The mathematical tokens for terms of the cost function are k instead of the intuitively chosen c. It is for a better distinction between terms concerning the cost function and processors for which the c already is used.

The notation of the following formulas is explained in chapter 3.1. In this work a definition of load for a processor is used, which also takes its power into account. The following term defines the load for a processor c:

$$c.\text{load} = \frac{c.\text{tasks}}{c.\text{power}} \tag{4.1}$$

It is the inverse of a processor's power normalised to the amount of tasks it runs.

The first term for the minimisation of the execution time of all tasks in the task set T of a program is a simple sum. It is the sum of all loads on the processors, to which the particular tasks of the program were assigned:

$$k_{\text{instr}} = \sum_{\forall t \in T} t.\text{cpu.load} \tag{4.2}$$

The application of this term during optimisation favours the assignment of tasks to powerful processors equipped with a lot of processing power, normalised to the number of tasks running on it.

The algorithm implemented in this work is additionally enhanced by weighting the originating load by the number of instructions the actual task consist of. Accordingly, the summed up term becomes: $t.\text{cpu.load} \cdot t.\text{instr}$. The consequence is that computational bigger tasks are mapped to processors having more power, with respect to the currently running tasks.

Another improvement can be done by weighting the term that is added up by the amount of programs that have running tasks on the processor to where task t is assigned. Using the previous formula, the resulting term becomes: $t.\text{cpu.load} \cdot t.\text{cpu.progs}$. Using this term results in minimisation of the amount of multiple programs on a processor.

It is desirable that the tasks running on one processor are part of as few different programs as possible. One major intention is to decrease the influence of one program on another program in case that static phase assumptions about it became invalid. For instance, when the execution of a program shows a different behaviour than it was specified by its software graph at static phase. E.g, some additional tasks appear, see figure 4.2. In this scenario, the load of the processor changes and all tasks on that processor are affected, regardless of the program they belong to. Another fact is that running many tasks of the same program on one processor potentially reduces the amount of communication costs among these tasks. Though, this

is less important for this term because there is another one that takes the communication costs into account.

Including both of the enhancements mentioned, the formula looks as follows:

$$k_{\text{instr}} = \sum_{\forall t \in T} t.\text{cpu.load} \cdot t.\text{cpu.progs} \cdot t.\text{instr} \tag{4.3}$$

The resulting term assures that bigger tasks tend to run on fast processors. Particularly on processors on which fewer other tasks are already running, or at least fewer tasks from other programs.

Another variant to include the enhancement terms would be to weight their influence. Doing so would result in more control over the additional terms. The following principle shows how this could be done by introducing the weights α, $\beta \in [0, 1]$, the influencing term i and the influenced plain term t:

$$\begin{aligned} \alpha \cdot i \cdot t + (1 - \alpha) \cdot t &= (1 + (i - 1)\alpha)t \\ &\approx (1 + \alpha i)t \end{aligned}$$

The latter approximation holds especially for big values of i. In the given case this results in the formula below:

$$\begin{aligned} k'_{instr} &= \sum_{\forall t \in T} (1 + \alpha \cdot t.\text{cpu.progs})\ t.\text{cpu.load} \\ k''_{instr} &= \sum_{\forall t \in T} (1 + \beta \cdot t.\text{instr})\ t.\text{cpu.load} \end{aligned}$$

In this formula, α weights the influence of the amount of programs on one processor: t.cpu.progs. In the same way, weight β affects the importance of the computational size of the tasks: t.instr. Taking both together, the resulting formula becomes:

$$k'''_{instr} = \sum_{\forall t \in T} (1 + \alpha \cdot t.\text{cpu.progs} + \beta \cdot t.\text{instr})\ t.\text{cpu.load}$$

This is a finer variant of the previous formulas. Using appropriate weights, the behaviour of the formulas above can still be achieved. Setting $\alpha = \beta = 0$ means no additional influence on the cost c'''_{instr}, as function 4.2. When setting $\alpha = \beta = 1$ the resulting term is similar to equation 4.3, but not equal. The variants with only α or $\beta = 1$ and the other zero results in the mentioned terms with only one influence.

The flexibility to be able to generate all conceivable weights comes with the issue that there are two new parameters, which have to be selected appropriately. This is the major drawback, because this can be pretty hard. The topic of finding suitable weights is presented in the next chapter 4.3.2.

The term for minimising the costs for communication of the program is the sum of the individual communication costs that arise using the actual mapping:

$$k_{\text{com}} = \sum_{\substack{\forall t,c \in T \\ t.\text{com}[c]>0}} \frac{t.\text{com}[c]}{\text{bandwidth}(t.\text{cpu} \rightarrow c.\text{cpu})} \; + \; \text{latency}(t.\text{cpu} \rightarrow c.\text{cpu})$$

This term simply sums up the communication times for every task to every other task it communicates with. It will ensure that tasks which exchange large amounts of communication data will be mapped onto the same processor, or at least to processors with a good connection.

The term for the minimisation of the costs for dependences is analogous:

$$k_{\text{dep}} = \sum_{\substack{\forall t,d \in T \\ t.\text{dep}[d]>0}} \frac{t.\text{dep}[d]}{\text{bandwidth}(t.\text{cpu} \rightarrow d.\text{cpu})} \; + \; \text{latency}(t.\text{cpu} \rightarrow d.\text{cpu})$$

It similarly sums up the times that tasks need to exchange their dependence data. Hence, tasks, which exchange a lot of dependence data will be mapped again onto the same processor respectively onto processors with a good connection.

The terms for dependence and communication costs could be combined into one term because of their similar structure. But that would bring the disadvantage of losing control over the specific aim in the whole optimisation formula. Introducing two different weights increases the complexity but allows a finer tuning of the cost function.

To support the distributed execution on multiple processors, the following term is used:

$$k_{\text{distr}} = \sum_{\substack{\forall t,c \in T, c \neq t \\ t.\text{com}[c]=0 \\ t.\text{dep}[c]=0}} \delta(c.\text{cpu}, t.\text{cpu}) \quad \text{with } \delta(c,t) = \begin{cases} 1 & c = t \\ 0 & c \neq t \end{cases}$$

This term favours the assignment of different tasks that have no communication or dependence relation onto different processors. In other words, it penalises the mapping of tasks not communicating with or depending on each other on one single processor.

There is an additional enhancement, which is realised in the implementation. The term is generalised for all tasks in the way such that all tasks, which share a processor get a penalty. This penalty must be small for tasks that communicate a lot, respectively that have a lot of dependencies. In contrast, the penalty must be high for tasks with low or no communication respective dependence. As formula:

$$k_{\text{distr}} = \sum_{\substack{\forall t,c \in T \\ c \neq t}} \left(1 - \frac{t.\text{com}[c] + t.\text{dep}[c]}{t^*.\text{com}[c^*] + t^*.\text{dep}[c^*]}\right) \delta(c.\text{cpu}, t.\text{cpu})$$

Note that tasks can either have a communication relation or a dependence relation, because communicating tasks run in parallel and dependent tasks run in sequence, as it was specified in chapter 3.2. In other words, having one relation implies not to have the other relation. In the formula, t^* and c^* denote the tasks for which the communication or dependence costs are maximal, depending on which ones is higher. In the latter formula, a penalty is generated for all tasks on a processor that has other tasks on it. This penalty is weighted inversely by the volume of the dependence data respectively the volume of the communication data. Using the last term, tasks will be mapped onto processors, which have no load at all at that moment. If that is not possible then they will be mapped onto processors that currently have a low amount of tasks assigned. Out of these processors, the processor is selected, which has tasks assigned that maintain strong relations to the given task (dependence or communication).

The resulting cost function that takes all terms into account becomes:

$$k_{\text{sum}} = w^{\text{sp}}_{\text{instr}} \frac{k_{\text{instr}}}{k_{\text{instr,max}}} + w^{\text{sp}}_{\text{com}} \frac{k_{\text{com}}}{k_{\text{com,max}}} + w^{\text{sp}}_{\text{dep}} \frac{k_{\text{dep}}}{k_{\text{dep,max}}} + w^{\text{sp}}_{\text{distr}} \frac{k_{\text{distr}}}{k_{\text{distr,max}}}$$

In this function, the $w_i^{\text{sp}}s$ are the weights for the terms, which are discussed in the next section 4.3.2. The superscript sp is to denote that the weights are for the cost function of the *s*tatic *p*hase in contrast to the weights in the dynamic phase, which will be discussed in section 4.4.2. The $k_{i,max}$ are used for normalisation to the range of $[0 \ldots 1]$ and represent the maxima of the particular term regarding all mapping combinations.

4.3.2 Weights - Parameters of the Cost Function

The identified weights have to be chosen with great care because they greatly influence the quality of the cost function as a whole. Some tests done using different weight combinations showed that appropriate weights are very difficult to select.

After having carried out several experiments, the following thesis could be made: the weights must be a factor that is proportional to the machine and to the program parameters. Parameters for a Machine are the load, the processing and the communication power. The parameters of software are the amount of instructions, dependences and communications. It turned out that there is a functional relation between good weights and the hard- and software parameters. Probably, the statistics of the parameters influence the functional relation, e.g. the proportion of parallel and serial tasks in a program. Likewise, the statistical measures such as maximum, mean and variance of tasks running in parallel could be important to the relation sought. For completeness: higher order statistics would be also interesting to investigate. For further details on statistics see [PTVF92].

In order to be able to find the relational connection, data samples consisting of good weights and the functional entities of the soft- and hardware is needed. In order to implement an automatic search for good weights to be able to derive this functional relation, an error measure of the cost function, which implies a measure for the weights, has to be defined. Such a measure determines how far the cost function is away from the simulated times. For a good cost function, the values of each pair of cost function values have the same relation as the appropriate measured values, e.g. smaller, bigger or equal.

To illustrate this, figure 4.4 presents for one program and several different mappings the simulated runtimes. The picture shows additionally the values of the cost function, one time with all weights equal to one, and one time with optimal weights. The values of the cost function are scaled down to the range of the simulated runtimes. This is done only to enhance the imagery. For the cost function are only the relations among each other important, not the absolute values. So the cost function should have the same shape as the simulated runtimes.

This means, that the cost function values plotted over the measured values should be a monotonic function in the optimal case. This representation is shown in figure 4.5 for the same data as shown in figure 4.4. It can be seen, that values of the cost function with optimal weights show a much lesser variance as the ones of the cost function with equal weights.

The following measurement is implemented: For each value in the cost function value series, the relation to every other cost function value must be compared. If it is the same as the relation of the corresponding measured value pair, a *no error counter* will be incremented otherwise an *error counter* will be incremented. The over all error is then the error-to-no-error ratio. Figure 4.6 shows the pseudo code of this `evaluateError()` procedure.

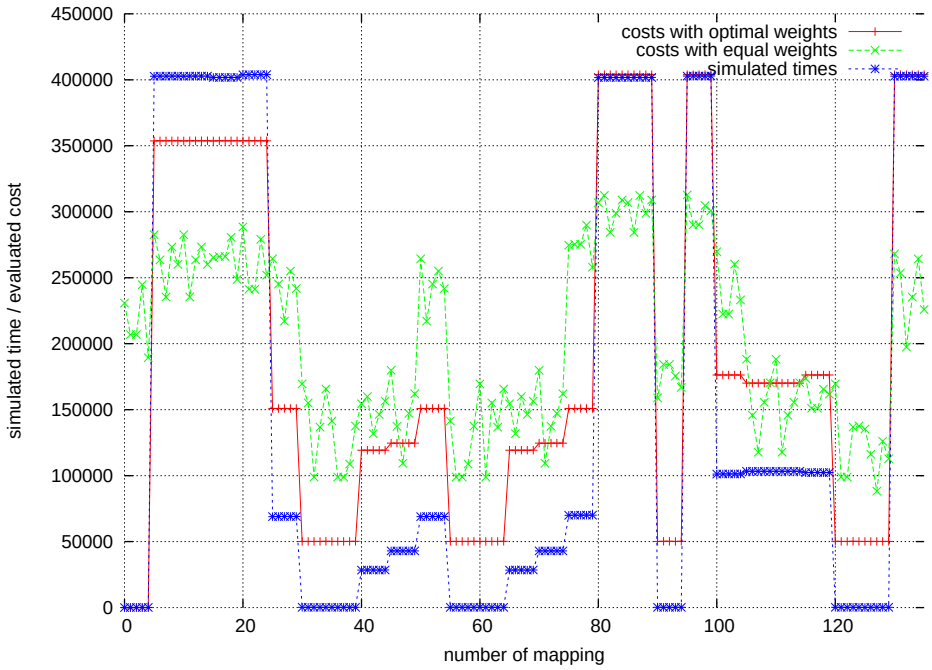


Figure 4.4: comparison of times and their corresponding cost function values with equal and optimal weights

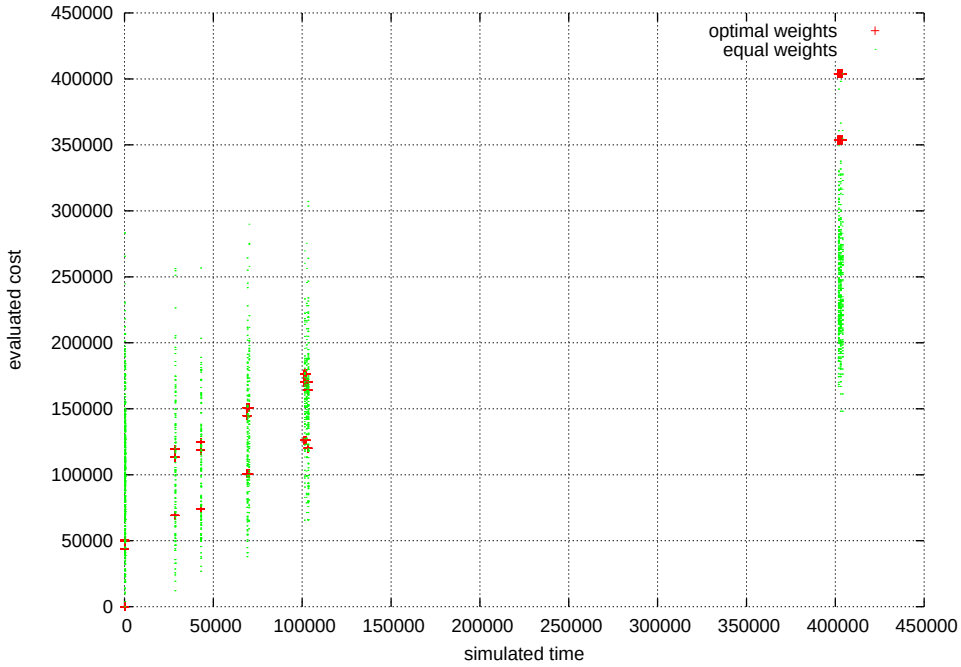


Figure 4.5: cost function values plotted over simulated runtimes

```
errorCounter ← noErrorCounter ← 0
cSeries[ ] ← generateCostFunctionValueSeries()
sSeries[ ] ← generateSimulatedFunctionValueSeries()
for i = 0 ... numberOfValues do
  for j = 0 ... i do
    if  (sSeries[i] < sSeries[j] ∧ cSeries[i] < cSeries[j])
     ∨ (sSeries[i] > sSeries[j] ∧ cSeries[i] > cSeries[j])  then
      noErrorCounter++
    else if  (sSeries[i] < sSeries[j] ∧ cSeries[i] > cSeries[j])
          ∨ (sSeries[i] > sSeries[j] ∧ cSeries[i] < cSeries[j])  then
      errorCounter++
    end if
  end for
end for
return errorCounter / noErrorCounter
```

Figure 4.6: evaluateError procedure as pseudo code for evaluating the quality of weights

The developed search algorithm as shown in figure 4.7 for two dimensions starts with all weights equal to 1, which is the lower left corner in the picture. It iterates over the weights until either no decrementation of the error measure in all loops of one iteration is detected or the step size, which decrements in every iteration, drops under a predefined threshold[5].

During each iteration the algorithm does the following. For every weight, the algorithm checks five values in combination to all the five values of the other weights [6] in a range around the initial value.

When a decrementation of the error measure is detected, the iteration resets and starts its loops again using the newly found weights. This results in more than five loops per weight, namely four steps farther the last found value. One iteration expressed in pseudo code is shown in figure 4.8. w_1 and w_2 are the initial weights, w_a and w_b are the temporary weights for the search around the initial ones, stepsize is the parameter which decreases from iteration to iteration and Δ is just a very small number that enforces the search never looks at a point again.

The algorithm quickly approaches the area of the global minimum of the optimal weights. Subsequently, it refines its search by decrementing the step size. Figure 4.9 shows the shape of the error measure along one weight. The outcome of the search shows that good results can be accomplished using the

[5] the predefined threshold is 0.0001 in the given case

[6] four weights result in a four fold loop with five checks each

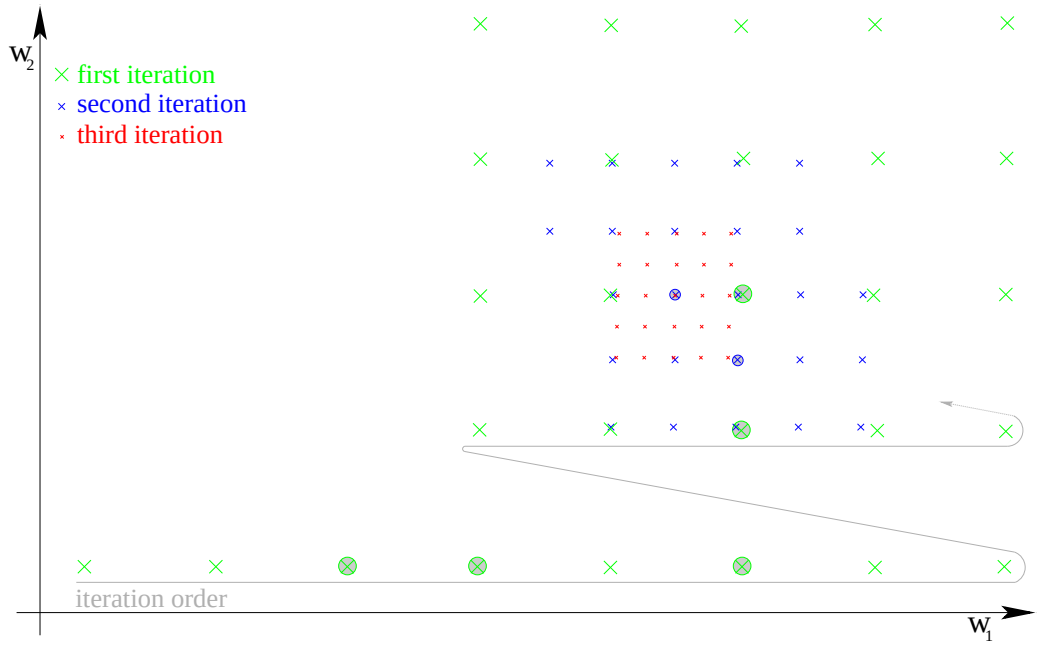


Figure 4.7: illustration of the search algorithm, encircled points denote the finding of a smaller value then the previous found minimum

$\text{error}_{\text{best}} \leftarrow \texttt{evaluateError}(w_1, w_2)$
for $w_a = (w_1 + \Delta - 2 \cdot \text{stepsize}) \ldots (w_1 + \Delta + 2 \cdot \text{stepsize})$ **do**
  for $w_b = (w_2 + \Delta - 2 \cdot \text{stepsize}) \ldots (w_2 + \Delta + 2 \cdot \text{stepsize})$ **do**
    $\text{error}_{\text{temp}} \leftarrow \texttt{evaluateError}(w_a, w_b)$
    if $\text{error}_{\text{temp}} < \text{error}_{\text{best}}$ **then**
      $\text{error}_{\text{best}} \leftarrow \text{error}_{\text{temp}}$
      $(w_1, w_2) \leftarrow (w_a, w_b)$
    end if
  end for
end for

Figure 4.8: pseudo code of one iteration of the weight search algorithm for two dimensions

method presented. The error is broadly a basin but with some zigzag lines.

With this search method, a broad relation was found, which is quite simple: The larger the sum of instructions is, a software has, the larger the $w^{\text{sp}}_{\text{distr}}$ weight ought to be. The same holds for the other weights. This rule corresponds to the presumptions, though, this result is unfortunately not

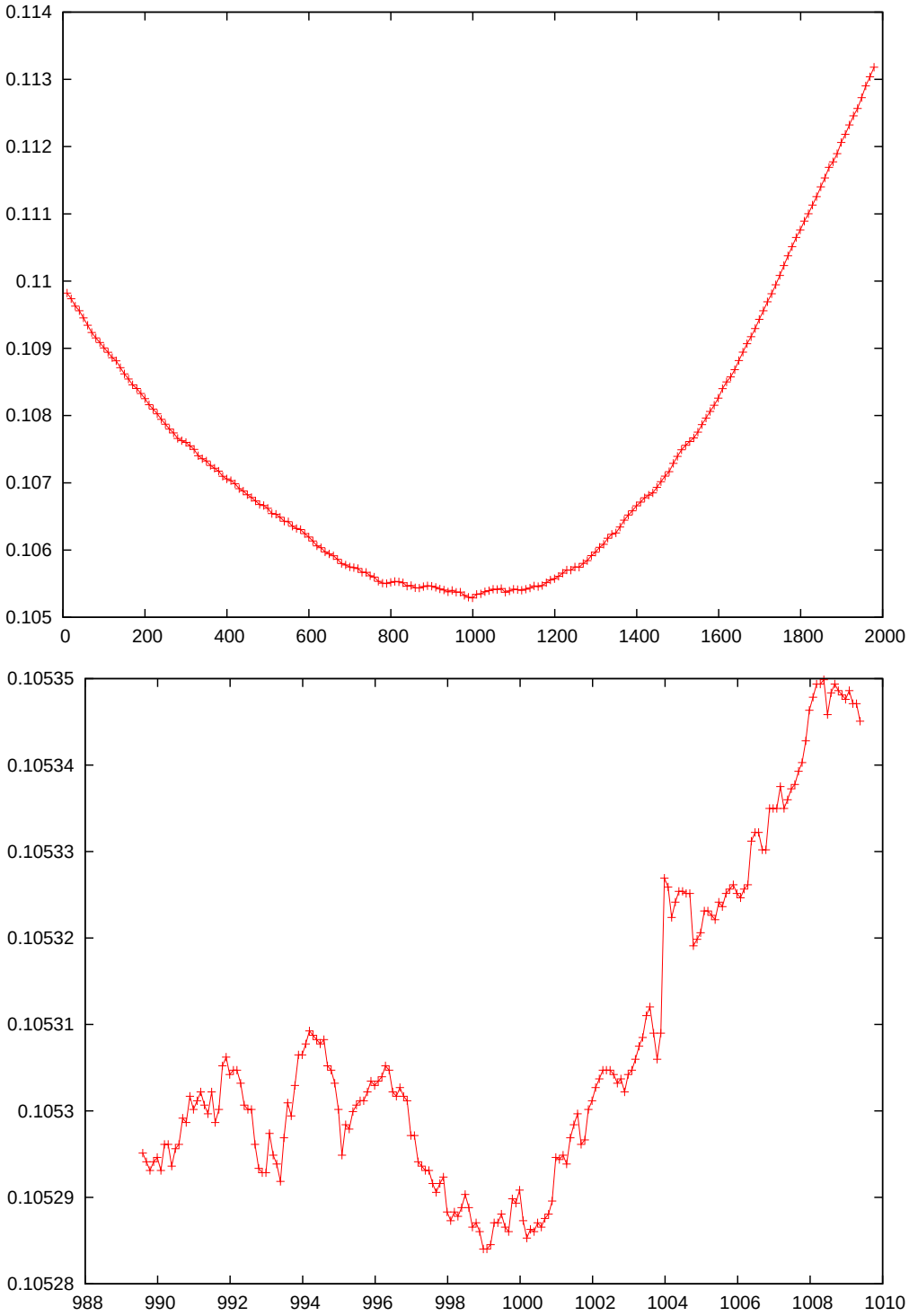


Figure 4.9: measured error along the weight dimension for the instruction term around the optimum found, above: in a wide range, below: in a small range

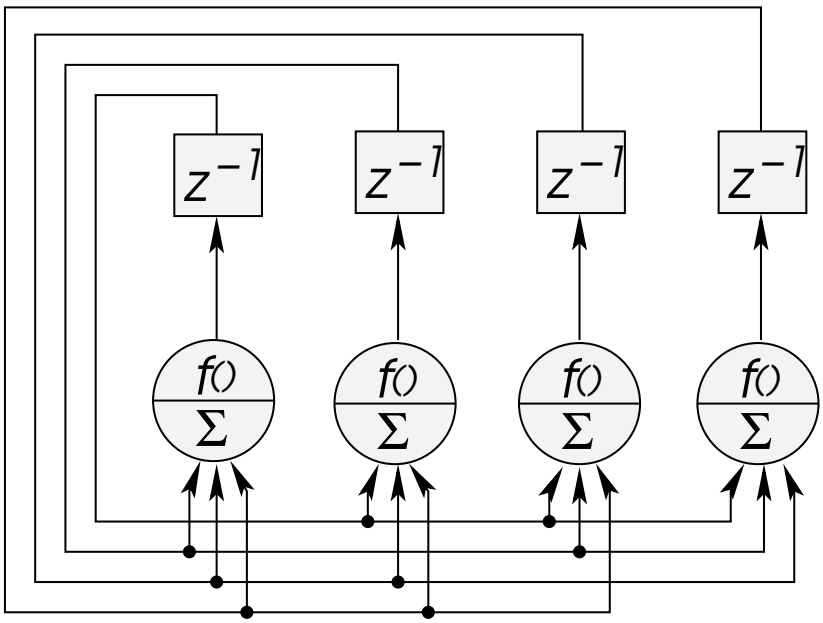


Figure 4.10: Neural Hopfield Network

universally valid. The weights used in this work are presented in chapter 5.1. Subsuming can be said, that the finding of good weights represents a task which is more complex and should be done in further work, see additionally chapter 6.1.

4.3.3 Hopfield Neural Networks

The Hopfield Neural Networks, originally discussed in [Hop82a] and [Hop82b], are one characteristic out of the whole class of neural networks and are often used for two different kinds of problem:

1. combinatorial optimisation problems as the Travelling Salesman Problem (TSP), see [Smi99], and

2. as content addressable memory (CAM), see [Hay99] and [Mac03].

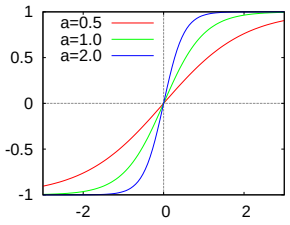
We deal with the former kind of network. Figure 4.10 shows the basic architecture of a Hopfield Neural Network. The circles are the N neurons, modelled after McCulloch and Pitts [MCP43], which sum up their input. This sum is called the activation a_k of neuron k on which it performs its sigmoid, non-linear activation function. After that it propagates the result as output with a time delay through a channel called dendrite to all other neurons. On the way to every other neuron the result passes a synapse in

the biological model. This synapse weights the output with the weight w_{ij}, from neuron j to neuron i, which is positive for an exitatoric synapse, or negative for an inhibitoric synapse. Additionally every neuron k receives a constant input, the bias b_k. As mathematical formula, the activation a_k and the output o_k of a neuron k can be expressed as follows:

$$a_k = \sum_{j=1}^{N} w_{jk} o_j + b_k \quad \text{and} \quad o_k = f(a_k)$$

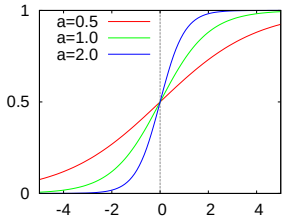
were $w_{ii} = 0$ to denote that no neuron propagates onto itself. Examples for the activation function are the continuous hyperbolic tangent function:

$$f_a(x) = \tanh(ax)$$



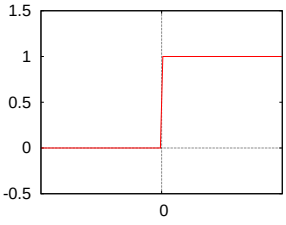
or the likewise continuous logistic function:

$$f_a(x) = \frac{1}{1+\exp(-ax)}$$



with a as slope parameter. In our case, where we deal with a combinatorial optimisation problem, we use the discrete threshold function:

$$f(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$



Depending on the activation function, a Hopfield neural network is called a discrete or a continuous network.

With this principle the network runs until it reaches a stable state where some neurons are always activated, the winners, which have an output near

the maximum of the corresponding activation function. And some neurons do not get activated any more, the losers, with an output near the minimum of the corresponding activation function. The running of the network minimises the energy function. It is shown for discrete networks:

$$E = -\frac{1}{2}\sum_{ij} w_{ij}a_i a_j - \sum_i b_i a_i$$

There are two things which have to be done to get a result:

1. building the network out of the cost function and inclusion of constraints [HKP91] and

2. run an algorithm on the network which iterates the network to a stable state, a minimum of the cost function.

In our case, every neuron represents the mapping of one task to one processor. So there are #tasks * #processors neurons. But in the set of winning neurons, which represent the solution mapping, must be only one neuron for each task. To enforce that, a constraint term will be added to the cost function:

$$k_{\text{con}} = \sum_i \left(1 - \sum_j o_{ij}\right)^2$$

To overcome more than one constraint there are some better approaches discussed in [Smi99] than adding several constraint terms to the cost function. The actual network, consisting of weights of connections and biases of neurons, are determined by comparison of the energy function of the network with the cost function of the given mapping problem. Thereby should match every term in the energy function one term in the cost function.

After the network is built an algorithm have to be chosen to run the network to an equilibrium. The many different algorithms can be categorised in synchronous and asynchronous update algorithms. Former algorithms update the outputs of all neurons at every iteration in one step. The latter algorithms choose every iteration one neuron and update its output. In [Ere02] a comparison of some selected algorithms is given for the task of graph matching with Hopfield Neural Networks.

The asynchronous *Steepest Descend Algorithm* is a simple greedy algorithm. In every iteration the neuron will be updated, which generates the maximal decrement of the energy function:

$$\Delta E_i = \min_j \Delta E_j < 0 \text{ with: } \Delta E_j = -(o_j(t) - o_j(t-1))a_j(t)$$

It terminates as soon as there are no more neurons which generate a decrement of the energy function. It starts from randomly initialised neurons. The algorithm looks as follows as pseudo code: This algorithm finds out-

```
random initialisation of states of neurons
while ∃i : ΔE_i < 0 do
  update the neuron with maximum energy difference
end while
```

going from the initialisation in a deterministic way a local minimum of the energy function. There will be no updates chosen which would increment the energy. So this algorithm is greatly endangered to walk into local minima.

The *Stochastic Steepest Descend* algorithm if very similar to the Steepest Descend algorithm. In the stochastic variant, the neuron which will be updated is randomly chosen from all neurons were an update would generate a decrement of the energy function. The prop ability $P_i(t)$ of neuron i to get updated at time t is proportional to the decrement that would occur:

$$P_i(t) = \begin{cases} 0 & \text{if } \Delta E_i \geq 0 \\ \frac{\Delta E_i}{\sum_{j \in C(t)} \Delta E_j} & \text{if } x < 0 \end{cases}$$

with $C(t) = \{k | \Delta E_k < 0\}$ the set of indices of neurons which update would generate a decrement of the energy. The algorithm looks as follows as pseudo code: The risk of running into a local minima is much smaller with this

```
random initialisation of states of neurons
while ∃i : ΔE_i < 0 do
  chose randomly a neuron to update with probability P_i(t)
end while
```

algorithm. The stochastic behaviour enables it to run through local barriers of the energy function.

Both variants of steepest descent, the stochastic and the deterministic, depend greatly on the initialisation of neurons, which is done by fortuity. So running the steepest descent from a bad origin will end in a not so good local minimum. To bypass this problem, the algorithm should run several times from different start points. In [Jag95] are as many cycles used as neurons are present, but in [Ere02] it was observed, that after 20-30 cycles, no significant changes occurred any more.

The *Mean Field Annealing* algorithm for continuous networks bases on the *Mean Field Theory* of the statistic mechanics. It is known to provide very good results but with the big disadvantage of needing very long to find

the solution. Because of this in combination with the need of fast responses of the desired mapping algorithm Mean Field Annealing was not selected for this work.

Most other algorithms suffer from the disadvantage to only be appropriate in some cases. The adaption would be possible but it would take much effort because of the many degrees of freedom of neural networks in general.

4.3.4 Simulated Annealing

As alternative approach came the *Simulated Annealing* algorithm into consideration. The quite simple algorithm bases on the annealing process, see [LA87] to get more details about the physical model. It is done to get straight crystalline structures. Thereby a solid object will be melted and than slowly cooled off. This process minimises the amount of free energy of the system. While the system is cooled down, it migrates from one state to another. The transitions to states with a lower energy level are always taken. But transitions to energetic unfavourable states are stochastic with a probability which depends on the temperature and the energy difference. So with the decreasing temperature descents the probability that the system transitions into states with higher energy levels, because of the decreasing amount of free energy. The temperature sinks so deep upon this process that in the end, the only transitions, which are possible are those leading into energetic favourable states. Eventually, the system will come to an equilibrium in an energetic minimal state.

In the Simulated Annealing algorithm this procedure is adapted and applied among others to combinatorial optimisation problems, see [Ere02], [KGV83], [Mac03] and [Hay99]. Given a combinatorial optimisation problem with a cost function f to minimise, the algorithm generates a new state s_{i+1} out of the current state s_i by doing an elemental step. In our case that would be to assign one task to another processor. The transition into the new state is stochastic with the following probability p:

$$p(s_i \rightarrow s_{i+1}) = \begin{cases} 1 & \text{if } \Delta f > 0 \\ e^{\frac{\Delta f}{T}} & \text{if } \Delta f \leq 0 \end{cases} \quad \text{with } \Delta f = f(s_i) - f(s_{i+1})$$

T is the temperature parameter which decreases in time and Δf is the amount of decrease of the cost function if the new state will be taken. The system transitions deterministic into this new state if the corresponding value of the cost function is lower than the one of the current state. If it is not lower, the occurrence of the transition is stochastic having the given probability. With the decreasing temperature the probability sinks that the system will migrate into states with bigger cost function values.

So the Simulated Annealing algorithm can be formulated in pseudo code as follows:

$s \leftarrow$ `randomStateInit`()
for $T = T_{max} \ldots T_{min}$ **do**
  $s_{new} \leftarrow$ `elementalStep`(s)
  $\Delta f \leftarrow f(s) - f(s_{new})$
  if $(\Delta f > 0) \vee$ (rand[0,1) $< e^{\frac{\Delta f}{T}}$) **then**
    $s \leftarrow s_{new}$
  end if
end for

The function `randomStateInit`() returns a state of the process which is chosen by random. T denotes the current, T_{max} the initial and T_{min} the termination temperature and plays the role of a control parameter. The function `elementalStep`() returns a state which differs from the given one in one elemental step. As mentioned above, that is in our case, that one task chosen by random is assigned to another processor, which is also chosen by random. Note that rand[0,1) just returns a uniform distributed random number in the interval from inclusive zero to exclusive one.

Up until here, there are still some things unclear. The first question is in which way the temperature decrements and which values are taken from T_{max} and T_{min}? In the literature are different possibilities to find to do the decrementation. In [Hay99] the following is stated:

> "The simulated annealing process will converge to a configuration of minimal energy provided that the temperature is decreased no faster than logarithmically. Unfortunately, such an annealing schedule is extremely slow - too slow to be of practical use. In practise, we must resort to a *finite-time approximation* of the asymptotic convergence of the algorithm. The price paid for the approximation is that the algorithm is no longer guaranteed to find a global minimum with probability 1. Nevertheless, the resulting approximate form of the algorithm is capable of producing near optimum solutions for many practical applications."

He further writes:

> "The annealing schedule due to Kirkpatrick et al. (1983) specifies the parameters of interest as follows:

Initial Value of the Temperature. The initial value T_0 of the temperature is chosen high enough to ensure that virtually all proposed transitions are accepted by the simulated annealing algorithm

Decrement of the Temperature. Ordinarily, the cooling is performed *exponentially*, and the changes made in the value of the temperature are small. In particular, the *decrement function* is defined by

$$T_k = \alpha T_{k-1}, \quad k = 1, 2, \ldots$$

where α is a constant smaller than, but close to, unity. Typical values of α lie between 0.8 and 0.99. At each temperature, enough transitions are attempted so that there are 10 *accepted* transitions per experiment on the average.

Final Value of the Temperature. The system is frozen and annealing stops if the desired number of acceptances is not achieved at three successive temperatures."

In the implementation of the GNU Scientific Library [GSL], as an example, is another variation of cooling schedule realised: a constant number of tries are taken at every temperature. And in [Ere02] the cooling schedule uses a constant number of updates, so the algorithm terminates with the reaching of this step count. During the first 25% of update steps the temperature sinks from 1 of 0.5, and for the last 75% it sinks to 0. The lowering is linear done.

In this work the cooling is performed in another way. The temperature is cooled down on every step as it is shown in the pseudo code above. The factor for cooling, α in [Hay99], is chosen in that way, that the number of steps is proportional to the product of the amount of processors and tasks. Additional it was realised, that quality of the solution depends on the initial state. To overcome this effect, the algorithm is run several times from different start points. Precisely, the algorithm starts #tasks + #processors times from random states and #processors times with all tasks on one processor.

In comparison to Hopfield Neural Networks, this algorithm turned out to be easier to control. One has to have the different concepts in mind: the cooling schedule in contrast to the neural networks. Using Hopfield Neural Networks, a constraint has to be included into the cost function and the activation function has to be chosen. Another important selection is the one of the algorithm for getting the result keeping in mind that the algorithm actually runs the network into its equilibrium.

Another advantage of the Simulated Annealing algorithm is the availability of a parallelised variant, e.g. in [MKBW92]. This could be interesting as this algorithm should run on cluster systems with many parallel nodes. But this is beyond the scope of this work.

4.3.5 Behaviour in Unforeseen Cases

The static mapping results in good performance in case the assumptions it made are not broken. The assumptions were:

- the load of processors a program is assigned to does not change due to other programs
- the program behaves as its software graph specified at the static mapping

In case one of the assumptions is broken, the quality of the static mapping may be reduced. One investigation that allows to validate the resulting behaviour is done as follows.

In this investigation, the mapping unit of the static phase works on different software graph as the one, which will be executed. The software graphs for this test, respective the scheme to build them, is depicted in figure 4.11. The mapping unit always gets the small basic graph, consisting of fife tasks. And the graph which will be executed is expanded with additional tasks. So only the basic tasks will be mapped by the static phase and all others will be detected as not mapped. As it is the native behaviour of the tasks, all of these unknown unmapped tasks will be assigned to that processor, where the first task terminates on which the unknown task depends.

In this investigation, the mapping unit of the static phase uses another software graph as the one that is being executed. The software graphs for this, the scheme to build them respectively, is depicted in figure 4.11. The mapping unit always gets the small basic graph consisting of five tasks. The graph, which will be executed is extended with additional tasks so that only the basic tasks will be mapped during the static phase. All others will be identified as not mapped. As a simple handling, all of these unknown, unmapped tasks will be assigned to the processor, on which the first task terminates the unknown task depends on.

The resulting mean runtimes of tasks are compared to the runtimes that the program needs when the mapping unit got the real software graph, which reflects the real program behaviour. The resulting mean runtimes and the acceleration that arise are shown in figure 4.12.

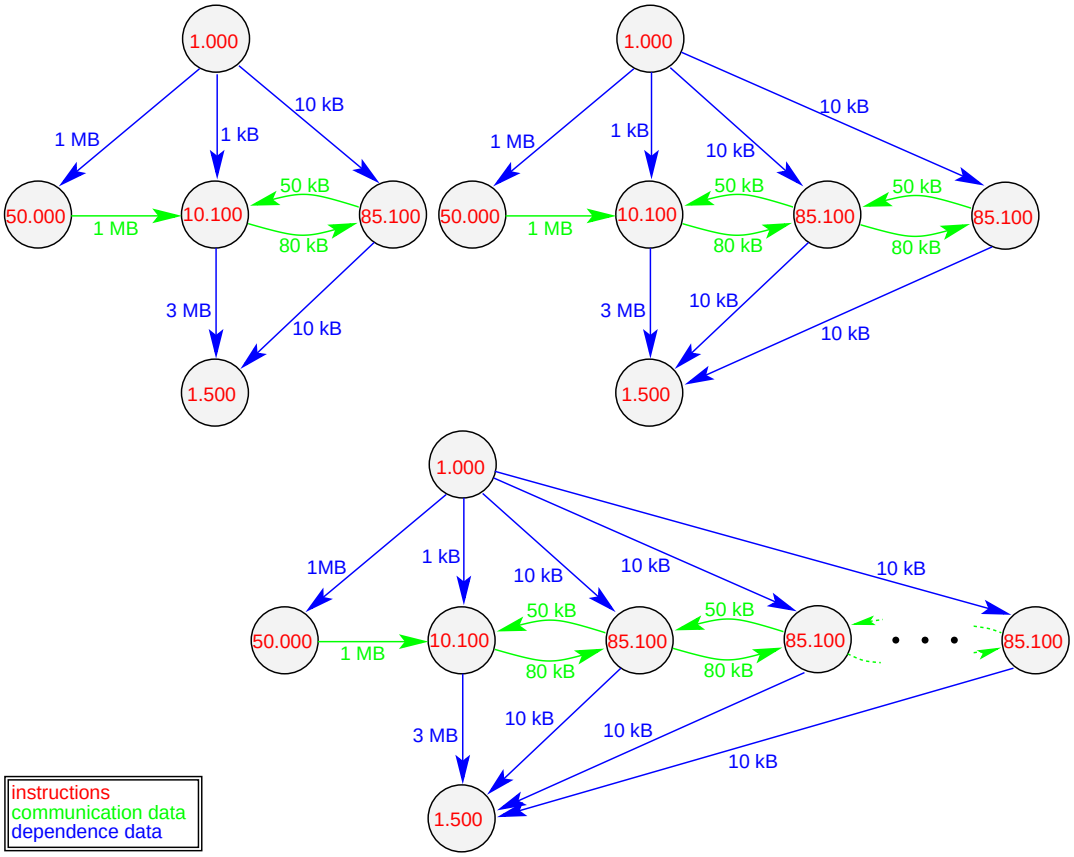


Figure 4.11: software graphs for testing, above left: graph known to the mapping unit, above right: 1^{st} extended executing software graph, below: principle of the extension

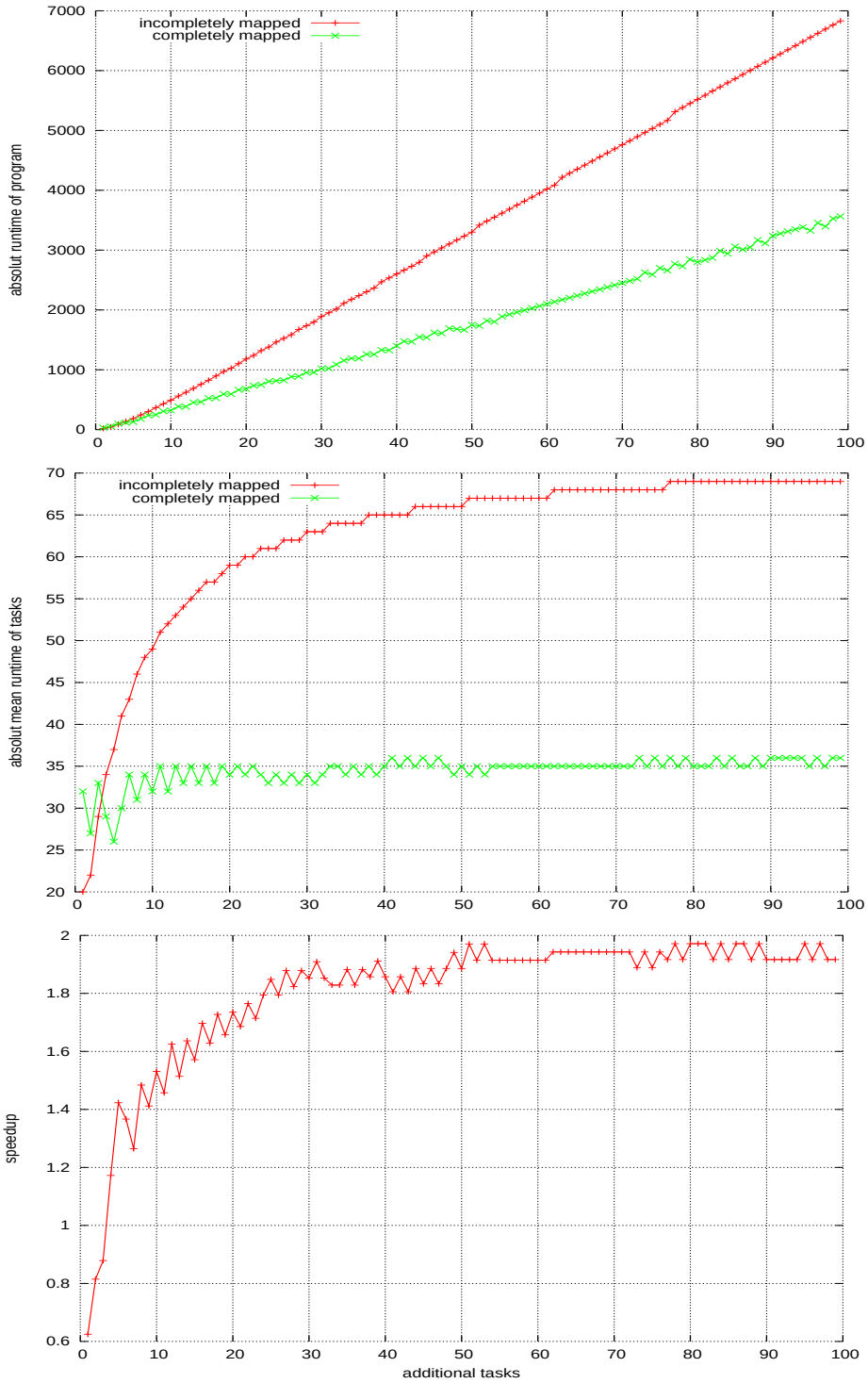


Figure 4.12: results of the tests with the graphs shown in figure 4.11, above: absolute runtimes of programs, middle: mean runtime of tasks, see definition on page 16, below: $\frac{\text{incompletelyMappedRuntime}}{\text{completelyMappedRuntime}}$ = ratio between runtimes, when all tasks were mapped and some were not, additional tasks on the x-axis means the amount of tasks which arise at execution, which were unknown to the mapping unit

This test was done with the same software graphs on different hardware graphs and the resulting measurements all look similar: The knowledge of all tasks lets the mapping unit find mappings which exploit the hole capacity of the machine, and not only the processors the originating tasks are running on[7]. So the response times of programs which were mapped completely have a less steep increase as their corresponding incompletely mapped programs where unknown, unmapped tasks arose.

There are two additional observations at the measured mean runtimes of tasks:

1. If there are only a little amount of unknown tasks, two in the shown example, the basic mapping results in better performance concerning the mean runtime of all tasks. But with more unknown tasks the resulting timings get worse in comparison to the timings, when all tasks were mapped.

2. The mean runtimes of tasks seem to go in a saturation with the increasing amount of additional unknown tasks.

The first observation is another indication for the suboptimality of the chosen algorithm in the static phase. It shows that there are better mappings than this algorithm has found. This was known, because of the complexity of the problem, the chosen algorithm generates only good mappings, not optimal ones.

The second observation, the saturation of mean runtimes of tasks, can be explained in the following way: The mean runtime of tasks goes with the increasing of the amount of tasks against the sequential runtime divided by the size of the used set of processors. This set consist only of one processor in the case of unmapped tasks. So the mean runtime goes against those of real sequential execution. In the case, that all tasks are mapped, the set of used processors is bigger. Consequentially the mean runtime goes against a lower value, which results from the economic usage of the assigned processors. In the letter case all processors are used, where the distributed execution is worthwhile. Hence, all processors are in this set, with which the costs for communications and dependences do not exceed the profit of the parallel execution.

The fraction between the runtimes, when all tasks are mapped and when some are not mapped is roughly a measure of the ratio of power the set of used processors are possessing. So the acceleration, when all tasks are

[7]That is, all processors will be used to where the communication and dependence costs won't be to high.

mapped, in contrast to the case with unmapped tasks, has its asymptotic upper border around the fraction between the power of the sets of used processors. But this theory underlies the assumption, that the runtime of software is mostly determined by the computation costs, and only in a small fraction by the communication or dependence costs. The bigger the costs for dependence and communication are, the more probable it is, that the actual tasks will run on the same processor. And with that, they will generate similar runtimes as the incomplete mapped program execution. But still in this case, the static mapping would assign the tasks to a processor with a lower load[8] if there are bigger differences between the processors. This can be different in the case of an incomplete mapping.

4.4 Dynamic Load Balancing

In order to overcome the poor performance of the static mapping, mechanisms for load balancing are introduced into the framework. The algorithm used for this task, the physical model of forces, descends from the field of distributed computers [Sch91], [Hei94]. Recently this method was applied successfully in a different field: UMTS and GSM cellphone networks [PDJM04]. It is used to balance the load between these two network types to allow a better exploitation of the specifics of the radio technologies with respect to coverage, capacity and services.

The following sections present the used algorithm for the task of load balancing. Initially the methods basic analogy in physics is shown. Subsequently the main algorithm as well as the adaption to the given environment is discussed. The chapter ends with the explanation of how the dynamic load balancing collaborates with the algorithms of the static mapping.

4.4.1 Physical Model of Forces

The algorithm used for load balancing at the dynamic phase was first discussed in [Sch91]. The idea is derived from a physical everyday phenomenon. Figure 4.13 outlines the experiment. In this experiment, several non-mixable fluids with different viscosities will be dumped into a basin. When doing this, the fluids will spread over the ground of the basin. It will be observable that some fluids will be distributed more balanced as others which is because of their different viscosities. Likewise, the surface will not be plain. When some new fluid is poured into the basin it will not be necessarily the only

[8] definition of load with respect to the power of a processor: equation 4.1, on page 24

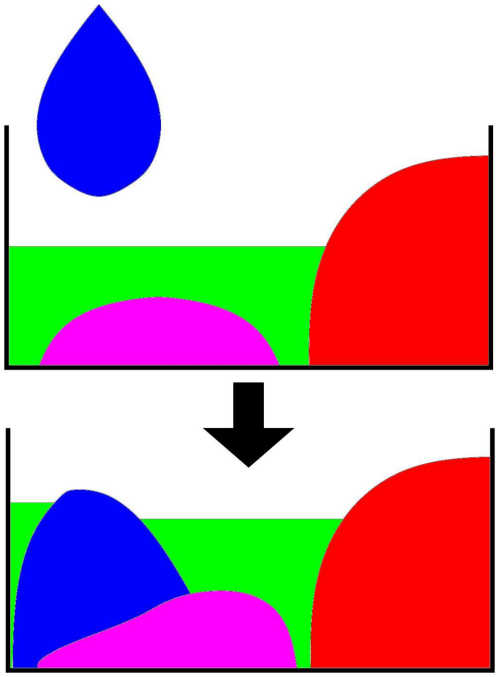


Figure 4.13: physical everyday experiment, the basic analogon to the dynamic load balancing algorithm

fluid, which distributes itself. Depending on the viscosities of the participating fluids it will also push away the fluids, which are currently at that place. Correspondingly, the surface may change its structure.

Load balancing is in principle a discrete analogon to this. Table 4.1 shows the counterparts of the physical model and the cluster system environment. In this analogon, the basin corresponds to a whole cluster computer, the computational power of a single processor to the capacity of an area in the basin and the liquid level complies to the load. Hence, the more power a processor posses, the more capacity contributes its corresponding area of the basin to the whole capacity. Reciprocally, the area provides as much area, capacity, as the amount of power of the corresponding processor inheres. A fluid in the physical model corresponds to a set of tasks of one parallel program. The amount of dependence and communication between the tasks correspond to the viscosity of the fluids.

At first the analogy differs by the fact that fluids can spread at almost arbitrary small amounts. Contrary, in this work, is a task in a cluster system the smallest entity, which can migrate from one processor to another. The second restriction is the fact, that the flow of fluids can go in nearly every direction, except through the walls of the basin. The ways of migration of

physical model	cluster system environment
basin	cluster computer
area in basin	processor
capacity of area in basin	power of processor
fluid	set of tasks of a program
viscosity	communication and dependence
foulness of basin at one point	load of processor

Table 4.1: correspondence between the physical model and the cluster system environment

tasks is limited to the network, the cluster system is equipped with. One can think of channels of different width between the locations of the fluids because of the fact that connections with more bandwidth permit the transmission of more tasks, respective their dependence data, from one processor to another one at the same time.

In physics, the behaviour of the liquids is modelled by effects of forces and potentials. Based on the analogon above, the same should be valid for tasks in a cluster system. The next section describes, which potentials and forces are taking effect and how they are defined.

4.4.2 Forces and Potentials

In the original work [Sch91], the author uses a quite different model. Load balancing is realised through migration of already running processes. A process migrates to the processor to which the strongest virtual force brings it. The forces are defined between the processor of the initiating process and all neighbouring processors. Each force is again a weighted sum of basic forces each of which aims for a particular goal, like small communication costs or fast response times. The basic forces are defined as differences of potentials, e.g. how much communication costs arise when the process runs on a certain processor? Such a cumulative force is defined for every neighbouring processor of the processor, the initiating process is assigned to. The biggest force virtually wins and determines the target processor for the migration.

The presented forces dealt only with the profit of migration, but there are also costs, which must be handled, e.g. the transmission time for the process description and its address space. To take this into account, two add-ons were introduced: One aspect is to reduce the amount of migratable processes in a stochastic manner. This is done using a uniform distributed random variable and one migration counter per process, which increments by one at every migration. Thus, a process is migratable if the ratio of this counter to the

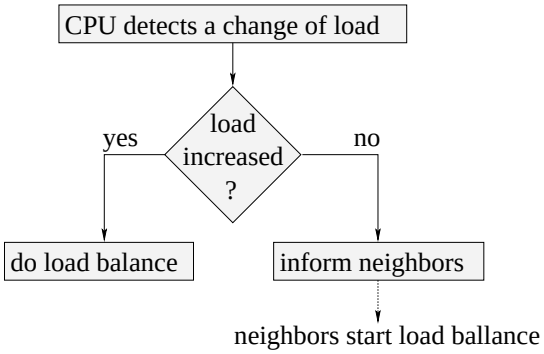


Figure 4.14: proceeding at start of load balance

maximal allowed number of migrations is smaller than a random sample. The second aspect is a threshold, which was added. This threshold depends on the costs, which arise through the migration, like the transmission of the process description as mentioned above. This threshold must be exceeded by the cumulated force, otherwise the corresponding processor will not be available as target for the migration.

Basically, load balancing should be done every time a change in the load is detected. But how is this concretely done? It is done using a decentral algorithm without any master-slave-relations. When a change of load is detected on a processor the load balancing is started, see figure 4.14. If this change is an increasing of load it will do load balance in the following way, see figure 4.15: The processor will subsequently detect all migratable processes on it, determine the maximal migration force for each migratable process, migrate the process according to the highest force and finally start the load balance again until the set of migratable processes is empty. Though, if the change of load is a decrease, the processor will simply instruct all neighbouring processors to do a load balance. In general, using this mechanism, the load balance can spread across the whole cluster system.

In contrast to the presented approach above, which uses forces and migration of running processes, this thesis is concerned with a similar problem, the problem of mapping small but not yet running tasks by means of virtual potentials.

When a task starts on a particular processor it gets bound to it and looses its ability to be remapped. This means that a task does not have an originating processor. It will start on the processor with the lowest potential. The potentials will be discussed below. As mentioned earlier in this thesis, the approximation to dynamic load balancing is the better, the smaller the tasks are.

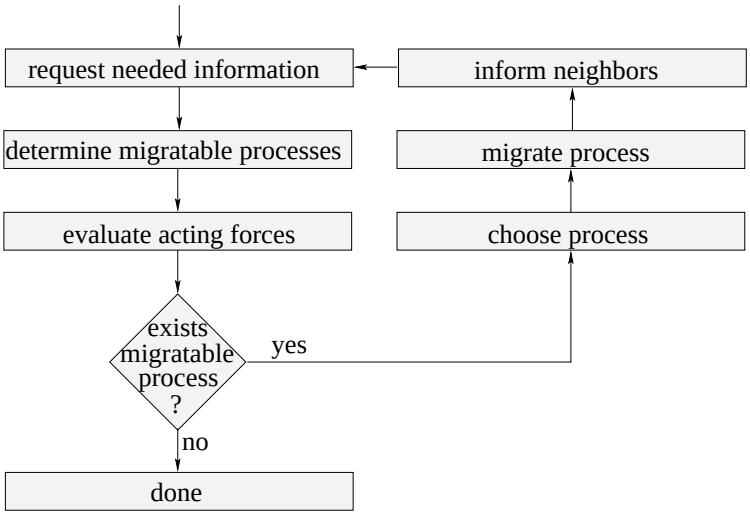


Figure 4.15: proceeding of one load balance step

First of all, the particular goals will be discussed, which will be approximated by the potentials. The goal of load balancing is often to minimise the mean response time. But there are other conceivable aims. If for instance the costs for communication are higher than for computation then one objective might be to keep the communication costs as low as possible. This would be a goal additional to the main goal of having small response times. Generally, three goals are identified.

- achieve balance of computational load
- minimisation of costs for communication
- minimisation of costs for dependence

The resulting potential function looks similar to the cost function of the static phase. It is also kind of a cost function that has to be minimised. It can be evaluated for every processor[9] and consists of a sum of weighted terms all of which represent a particular goal. The main difference in comparison to the static phase cost function is that the terms of the static phase consist of a sum over all task. Contrary the ones of the dynamic phase consider only one task.

The first term addresses the goal to balance the computational load. First, we have to have a definition of load. Analogous to the cost function

[9]the evaluation of the potential function for one processor corresponds to one assignment

in the static phase and with respect to the inhomogeneity of the processors, the load of a processor c is defined as follows:

$$c.\text{load} = \frac{c.\text{tasks}}{c.\text{power}}$$

The computational potential for a particular mapping of a task t is defined in this work as just the load of that processor:

$$p_{\text{instr}} = t.\text{cpu.load}$$

This simple measure can be extended likewise the corresponding term in the static phase cost function has been extended. It could be weighted with the amount of programs which are already running on the processor in question, or the influence can be weighted. The weighting of the term with the amount of instructions of the task does not make sense because it is constant for all mappings of the given task. So the amount of instructions has no influence on the dynamic mapping.

The second term is used for the minimisation of communication costs. It sums up the costs for all communications, the starting task would generate on a particular processor. This again corresponds pretty much to the cost function used in the static phase.

$$p_{\text{com}} = \sum_{\substack{\forall c \in T \\ t.\text{com}[c]>0}} \frac{t.\text{com}[c]}{\text{bandwidth}(t.\text{cpu} \rightarrow c.\text{cpu})} + \text{latency}(t.\text{cpu} \rightarrow c.\text{cpu})$$

This term makes sure that a new starting task will not be mapped on a machine too far away from its communication partners. In particular, tasks that have strong communication relationships will be close, preferably on the same processor.

The term for the dependence potential is defined analogous:

$$p_{\text{dep}} = \sum_{\substack{\forall d \in T \\ t.\text{dep}[d]>0}} \frac{t.\text{dep}[d]}{\text{bandwidth}(t.\text{cpu} \rightarrow d.\text{cpu})} + \text{latency}(t.\text{cpu} \rightarrow d.\text{cpu})$$

It sums up all dependence costs that arise with the mapping of t to the particular processor. Correspondingly this term makes sure that the given task will be mapped near tasks to which it maintains big dependence relations or vise versa.

A term to ensure the stability of the method is not needed. As mentioned further up, the original version of this method in [Sch91] included multiple

migrations of processes while they where running. So it had to be ensured that processes did not migrate back and forth. This can not happen in the given case, because every task will be mapped dynamically at most once. So the occurrence of such "ping pong effects" is not possible.

The resulting potential function which combines all the terms above is as follows:

$$p_{\text{sum}} = w^{\text{dp}}_{\text{instr}}\ p_{\text{instr}} + w^{\text{dp}}_{\text{com}}\ p_{\text{com}} + w^{\text{dp}}_{\text{dep}}\ p_{\text{dep}}$$

It is simply the weighted sum of all terms. Its result depends on the chosen weight parameters and the given task and processor for which the potential should be evaluated, $p_{\text{sum}} = p_{\text{sum}}(\text{task, cpu})$. So there are three parameters, $w^{\text{dp}}_{\text{instr}}$, $w^{\text{dp}}_{\text{com}}$ and $w^{\text{dp}}_{\text{dep}}$, which has to be chosen. The superscript dp is just to denote, that the weights are for the potential function of the *d*ynamic *p*hase in contrast to the weights of the cost function of the static phase.

4.4.3 Algorithm

The dynamical phase will be entered when a task is starting. Additional dynamic mapping needs to be done when the assumptions made at the static phase are mismatched. There are four events triggering.

too-big-load-event the load of the processor, the starting task was assigned to, is bigger than it was expected

too-low-load-event the load of the processor, the starting task was assigned to, is lesser than it was expected

unknown-task-event an unmapped task that was unknown to the static mapping unit, has been detected

missing-task-event the experience of the lacking of a task that was mapped during the static phase

The first two events may occur when the cluster machine is operated in *time sharing mode*. In this case, it is possible, that two or more programs start with overlapping runtimes and also with overlapping partitions[10] to which their tasks are assigned to. The first too-big-load-event will be triggered in the following case: a task of the previously started program starts a task on a processor while the latter started program has already running tasks on that processor. In this situation, a load increase will be detected. The too-low-load-event may arise when the second program starts a task on a processor, on which a first program has finished its tasks, but after the start

[10] partition - set of processors

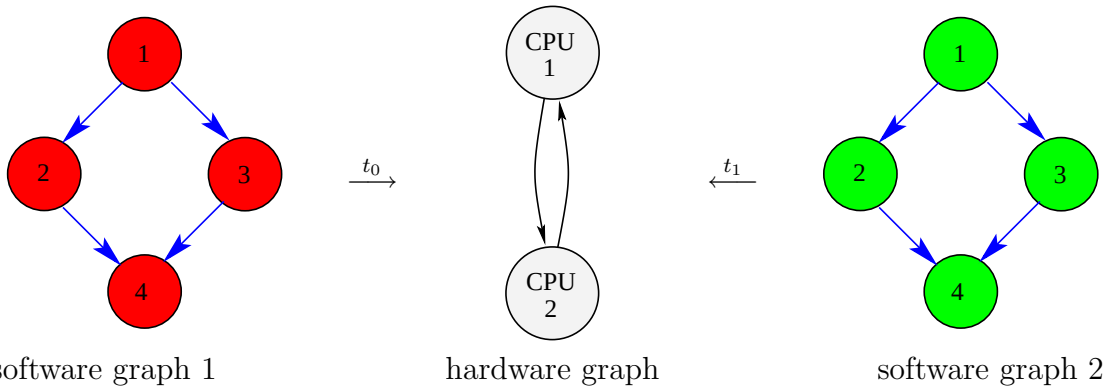


Figure 4.16: soft- and hardware graphs to illustrate the too-big-load- and too-low-load-event

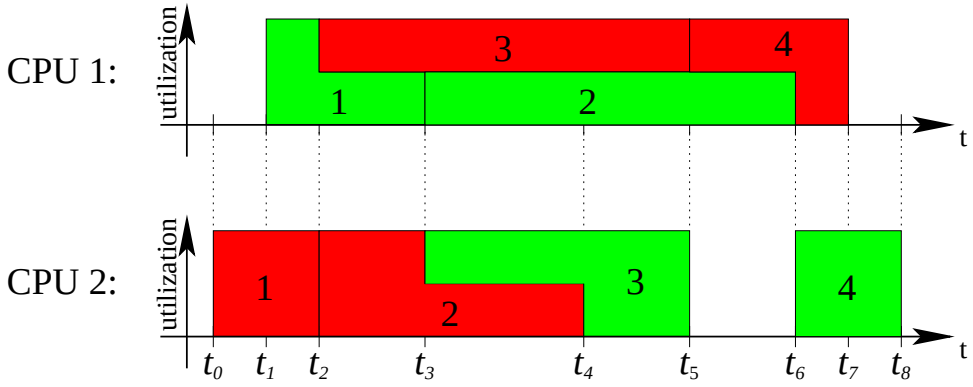


Figure 4.17: possible execution of the software graphs on the hardware

of the second program. In both cases, the load of a processor, to which a just starting task was assigned, changed. The change is caused by another program, not by the one that just starts a task.

In figure 4.16, 4.17 and 4.19, a simple example is shown to illustrate these two events. In figure 4.17 a possible execution of the software graphs on the hardware depicted in figure 4.16 is shown. Dynamic remapping will be triggered at t_2 and t_4 since the load produced by red task 3 respective 4 leads to a too-big-load-event. A too-low-load-event will occur at t_6 when the green task 4 starts and finds an empty processor 2.

The latter two events, unknown-task-event and missing-task-event, will be triggered every time when a program does not behave according to its software graph that was specified and used during the static phase map-

ping. The handled cases are those, in which a task that was specified in the predicted software graph does not appear during execution time. The inverse case is also handled, the case when a task is starting at execution time, which was not known at the static phase. Cases, in which additional or missing communications or dependences occur are not handled. This could be done in future work. Though, in order to investigate such circumstances correctly, a simulator might be needed that has a more complex model of network usage.

How the upper two cases are handled is nearly equal to how they are handled using the original algorithm [Sch91], depicted in figure 4.14.

In the given case, the handling of the too-big-load-event and unknown-task-event is done almost identical: a load balance is done for the task just starting. For this, the physical model of forces is used. This means that the potential function will be evaluated for every combination of the given task and the processors, which are the candidates for migration. The processor having the lowest potential will be chosen as the target for migration. In pseudo code, this procedure for a task t looks as follows:

$\text{potential}_{\text{best}} \leftarrow \infty$
migrationCPUs $\leftarrow$ `cpuCandidatesForMigration`(t)
for all $\text{cpu}_{\text{temp}} \in$ migrationCPUs **do**
  $\text{potential}_{\text{temp}} \leftarrow p_{\text{sum}}(t, \text{cpu}_{\text{temp}})$
  if $\text{potential}_{\text{temp}} < \text{potential}_{\text{best}}$ **then**
    $\text{potential}_{\text{best}} \leftarrow \text{potential}_{\text{temp}}$
    $\text{cpu}_{\text{best}} \leftarrow \text{cpu}_{\text{temp}}$
  end if
end for
return cpu_{best}

In this snippet of pseudo code, the procedure `cpuCandidatesForMigration()` will return a set of processors that contains all processors, which are favourable for the task to be migrated to.

The difference in handling of the too-big-load-event and the unknown-task-event is in the set of processors that the algorithm considers containing candidates for the migration. The question arises which processors are considered as target for the mapping.

The set of processors should not be to large since this would slow down the execution. Thus, it can not be the set of all processors of the cluster computer. In case of the unknown-task-event, the set consists of all processors that run all already mapped tasks that have a relationship to the task of interest. The direction of the relationship is irrelevant. Additionally, the

set contains all direct neighbours of the already identified processors.

In Figure 4.18 is an example presented. Shown is a hardware graph of a cluster computer running the software. Task 3 is currently dynamically mapped using the physical model of forces. The red nodes of the hardware are in the set of processors that the dynamic mapping will use for task assignment. The pink task 2 is known now but still unmapped. Hence, it does not influence the mapping of task 3.

The pseudo code for the procedure `cpuCandidatesForMigration()` that collects all these processors for an unmapped just starting task t of task set T is:

migrationCPUs $\leftarrow \emptyset$
for all $n \in T$ **do**
  if n.mapped = true $\wedge$ t.com$[n] > 0 \vee t$.dep$[n] > 0$ **then**
    migrationCPUs $\leftarrow$ migrationCPUs $\cup$ n.cpu $\cup$ `neighbours`(n.cpu)
  end if
end for
return migrationCPUs

The function `neighbours`() returns all direct neighbours of the given processor. These are the processors having a direct connection, not the processors that can only be reached using messages.

In the too-big-load-event, the set of processors will be collected the same way. Additionally, the processor and all its direct neighbours of the starting task itself will be added.

The too-low-load-event and the missing-task-event are handled similar to how the too-big-load-event and the unknown-task-event are handled. In case of lower load on a processor or in a case that a task does not start[11], the possibility that other tasks can run on the concerning processor without any disadvantage becomes higher. In each of these cases, the neighbouring processors will be informed about the now reduced load. The receiving processor maintains a list for such events and appends the ID of the sending processor to that list. The next time, one of the receiving processors starts a new task, it uses the potential function p to check whether it is appropriate to migrate the newly starting task to the processor with reduced load.

The too-low-load-event and the missing-task-event do not really triggers the dynamic mapping. They potentially trigger the load balancing with a delay. The delay, with which the next task starts on a neighbouring processor. The real trigger is the presence of processor IDs in the list of processors having reduced load while a task is starting.

[11] which means that lower load will be generated on the processor it was assigned to

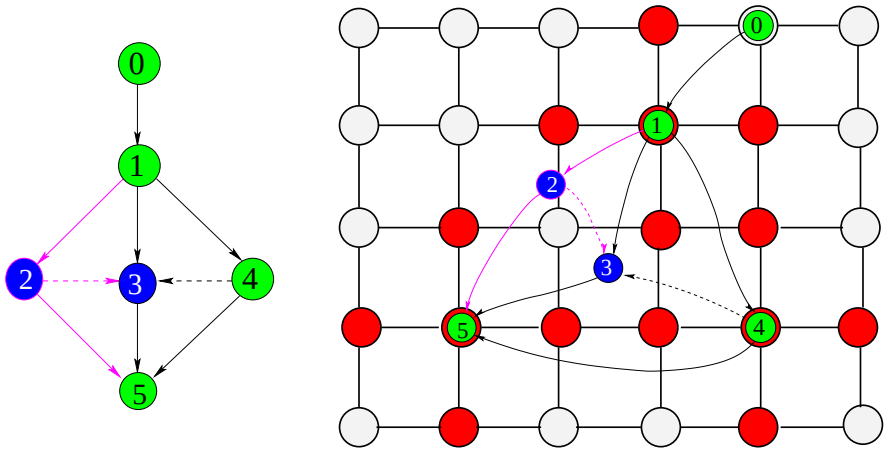


Figure 4.18: left: software graph showing mapped (green) tasks and unmapped (blue) tasks, communication (dotted) and dependence (drawn through) relations right: hardware graph of the cluster computer running the software

Every processor has to maintain a list of processors that currently have reduced load. After every notification about another processor with reduced load that a processor receives from a neighbour, it will append the sender's ID to this list. And every time a task is about to start on a processor with a non-empty list, this processor and all the processors of the list will be tested with the potential function p and the starting task. The task will start on the processor with the lowest potential. The list will be maintained as mentioned above. In the case, when the result of the potential function turns out, that the current processor is the best one for the task, the task starts on processor it was assigned to, it does not migrate. In this case it is assumed, that there is no unbalance any more and the whole list will be deleted. So no more dynamic remapping will be triggered until new entries will be inserted into the list, or some software does not behave as its graph has specified.

Another aspect is the load a program assumes for every processor its tasks are assigned to. This is important for the too-big-load- and too-low-load-event, which are triggered, when the load of a processor is not as it should be. To realise this, a list will be generated for every program at its start. These lists consist of the loads of the processors its tasks are assigned to. In this case is a simple counter for tasks for every processor sufficient. Initially, before any task of the just starting program has started, this task counters are, equal to the amount of running tasks on the processors. It will be incremented by one on every start of a task of the same program,

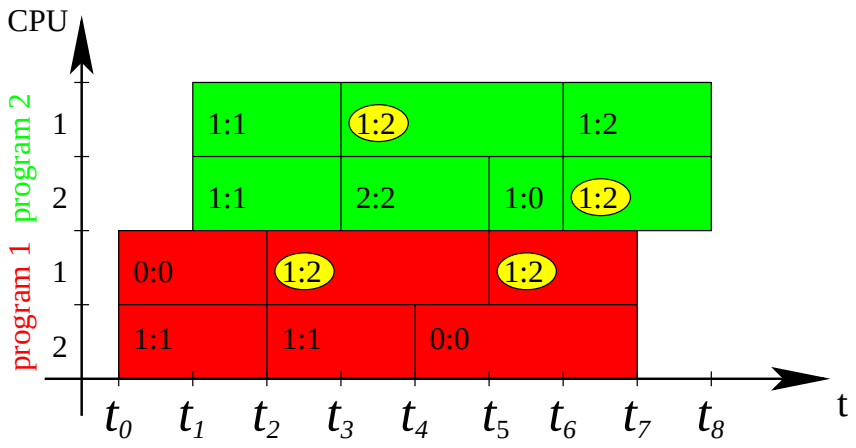


Figure 4.19: the data structures and its handling for the executions of both programs shown in figure 4.17

and decremented on every finishing. Additionally, at every start of a task, it will be checked, if the load of the particular processor has changed, that means, if the amount of running tasks differs from the value maintained for that program and processor. If it has changed, then the dynamic remapping takes place, as mentioned above. Otherwise the task just starts.

The values of this list of task counters per processor have to be updated at every start and end of tasks of the same program. For example, if a task of a program starts on a processor, where another task of the same program will start a moment later, then the later starting task should not trigger the dynamic load balancing, because the static mapping had that under consideration. But if the two tasks are from different programs, the dynamic remapping should take place, because this case is beyond the scope of the static mappings which had taken place.

An illustrating example is shown in figure 4.19 together with 4.16 and 4.17. It shows the data structures and its handling for the executions, figure 4.17, of both programs shown in figure 4.16. The variation of the maintained task counters (left) is shown over the time and compared to the real amount of tasks (right) for each processor of each program. These value pairs are updated always directly after the start or finish of a task. If these values differ at start of a task, then the dynamic remapping will be triggered. These cases are yellow encircled. If they differ at the ending of a task will nothing happen.

5 Practical Issues & Evaluation

The two tasks, static mapping and dynamic load balancing, can not be divided into two independent units. The mapping unit could run standalone. Because the dynamic load balancing is rather an enhancement, it is for the handling of the mentioned cases which exceed the scope of the static mapping. It uses data which bases on the static mapping and was generated by the mapping unit. So the two tasks are combined to one slightly more complex framework.

This chapter elaborates on practical issues and evaluations. Therefore the first section is mainly subjected to the theme of finding weights for the cost and potential function. The following section presents the experiences with algorithms used in the static phase. And in the last two sections are simulation results presented and evaluations made.

5.1 Weights

In chapter 4.3.2 an automated exhaustive method was presented to find optimal weights for the cost function of the static mapping. This was done to find a rule to determine the weights out of the parameters for the hard- and software descriptions. During that search it turned out, that the weight k_{distr}, which should favour the distributed execution of several tasks is probably unnecessary. In all runs of the weight search, this weight was near zero and much smaller as the other weights. The simple cross validation test, setting k_{distr} to zero, did not influence the results significantly. So the whole term for the distributed execution can be left out.

The weight k_{distr} was adapted from [Sch91]. The parallel execution of tasks of programs is not a direct goal of this framework, but a means to speed-up the response times respective scale-up the problem to solve. Consequently this result is not surprising. In consideration of the enormous influence of weights onto the goodness of the resulting cost function, this omitting simplifies the problem greatly. Additionally, the cost function looses some of its computational complexity.

In contrast to this result, the finding of a rule to determine the weights is much harder. A broad result of the simulations is: The weight k_{instr} should be bigger, if the amount of instructions in the software graph is also bigger. The analogue holds for the weights k_{com} and k_{dep}. For the examined examples, the following calculations resulted in a good cost function for a program consisting of a task set T:

$$\begin{aligned} w_{\text{instr}}^{\text{sp}} &= \frac{1}{3.5} \sum_{t \in T} t.\text{instr} \\ w_{\text{com}}^{\text{sp}} &= \sum_{t,c \in T} t.\text{com}[c] \\ w_{\text{dep}}^{\text{sp}} &= \sum_{t,d \in T} t.\text{dep}[d] \end{aligned}$$

So $w_{\text{instr}}^{\text{sp}}$ is roughly 0.28571 times the sum of the instructions over all tasks of the software graph. Similarly, without any factor, $w_{\text{com}}^{\text{sp}}$ and $w_{\text{dep}}^{\text{sp}}$ are the sum of communication respective dependence data the tasks possess.

Unfortunately, this relation of weights is not universally valid and the weights result only in a good cost function for a limited set of test graphs. The tests were done with software graphs of the form shown in figure 3.3, all values were set to small values. In every run, one value, e.g. the amount of instructions of the first task or the amount of communication data task 2 sends task 3, where varied. For these cases the found calculation of the weights are mostly sufficient, shown in figure 5.1. This figure illustrates, that this calculation fits fine for the variation of the communication values of the test graphs. It also fits a bit worse, but still sufficient to the variation of the dependence values. But this calculus looses its quality at the variation of the instruction values. With this, only about $\frac{3}{4}$ of the generated results will be good. Shown in figure 5.2, all values are set by random. In this case the results are really poor.

A problem which arose at the search is the complexity of the task of the search. To achieve statistical significance, a lot of different software graphs should be tested on many different hardware graphs. But in every test the runtime have to be generated for every possible mapping. This already takes a long time. Additionally there have to run the automated exhaustive weight search algorithm. This takes even more time. As an example, all tests with a five-task-software graph and a five-processor-hardware graph have taken much more than one hour. Some tests with this configuration took about ten hours and longer. The tests were done on the authors PC, an Athlon 1800 XP respective similar computers.

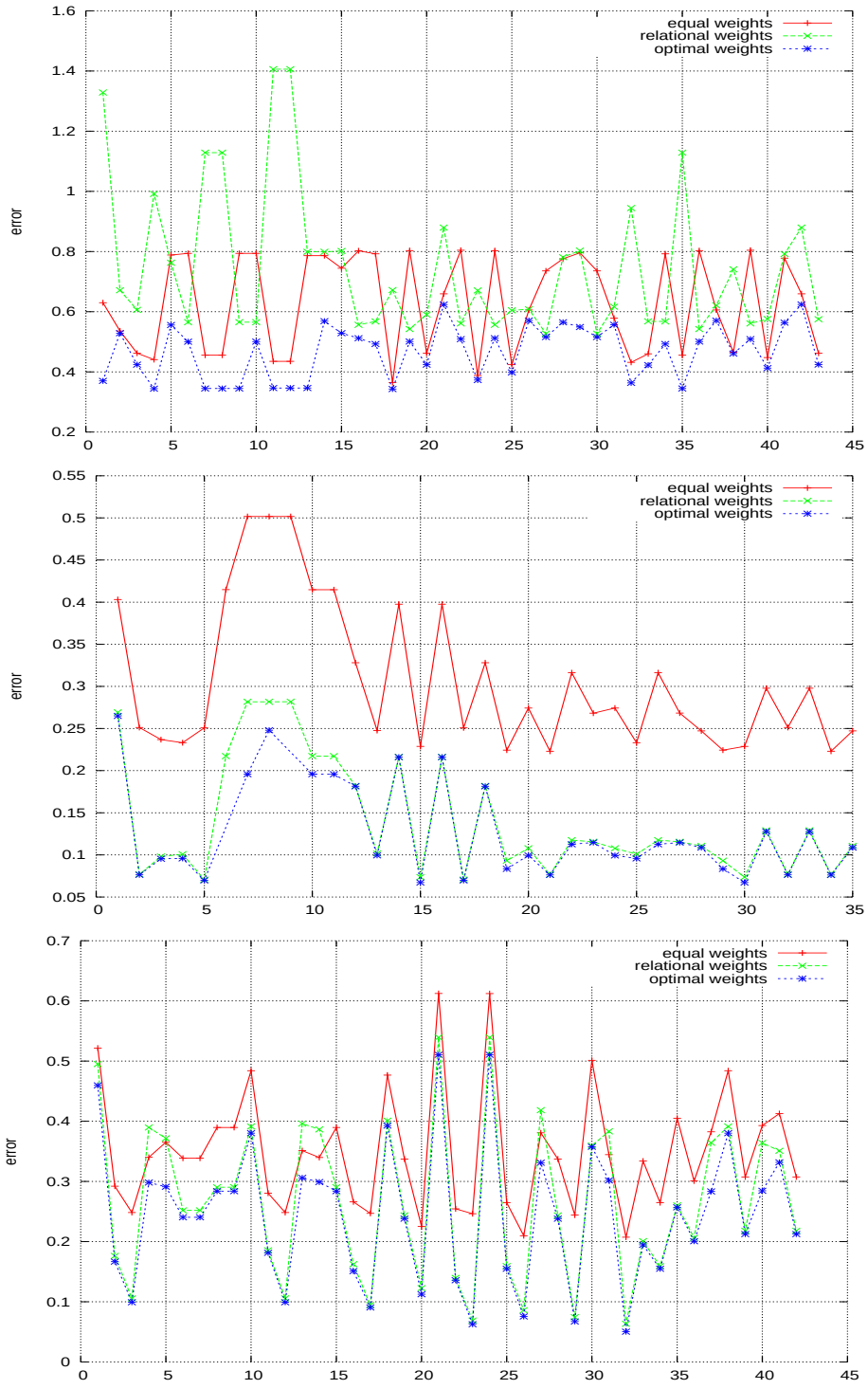


Figure 5.1: error of the cost function in contrast to the simulated runtimes of the software graph depicted in figure 3.3 with varied values; above: variation of the instruction values; middle: variation of the communication values; below: variation of the dependence values;

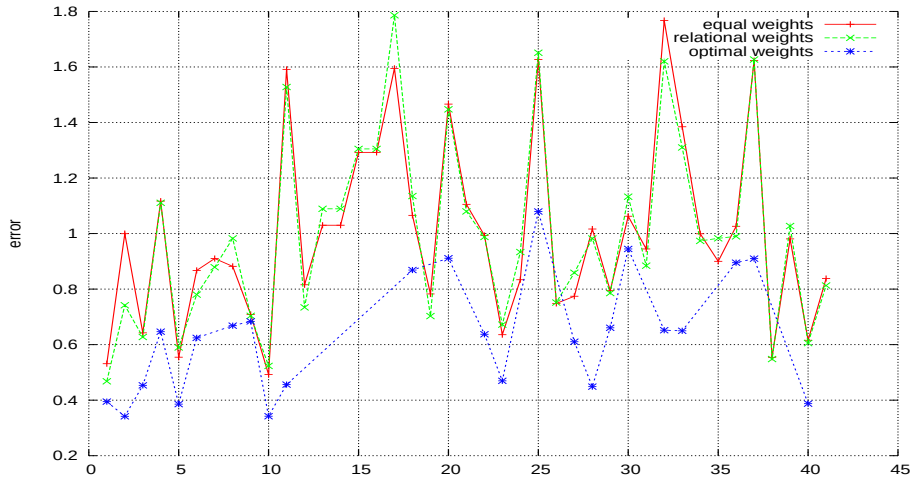


Figure 5.2: error of the cost function in contrast to the simulated runtimes of the software graph depicted in figure 3.3 with randomly varied values; some tests had to be aborted because of there long runtime, that's why there are some missing points in the series for optimal weights

The weight for the constraint of the Hopfield Neural Network should not be smaller as any other weight. Its simply because of the reason, that it stands for the most important goal, the validity of the result. But it should be as small as possible, so that it does not influence the results. In this work, a suitably weight was generated by taking a value around 1.8 times of the biggest weight of the normal cost function.

The weights of dynamic load balancing, $w^{\mathrm{dp}}_{\mathrm{instr}}$, $w^{\mathrm{dp}}_{\mathrm{com}}$ and $w^{\mathrm{dp}}_{\mathrm{dep}}$, are not as sensible as their counter parts of the static phase. With some simple experiments were some different weights tried out. The result showed that the following values were quite good for different software and hardware graphs:

$$w^{\mathrm{dp}}_{\mathrm{instr}} = 800 \quad \& \quad w^{\mathrm{dp}}_{\mathrm{com}} = 1 \quad \& \quad w^{\mathrm{dp}}_{\mathrm{dep}} = 1$$

Experiments showed that the weights for communication and dependence terms in the static and dynamic phase are similar in most experiments. So maybe, this terms can be integrated into one common term for communication and dependence. With this the ability of changing the weights independently would be lost. But the advantage would be to reduce the number of weights, which is favourable in order to reduce the complexity to find suitable weights.

5.2 Hopfield Neural Network versus Simulated Annealing

As outlined in [HKP91], [Hay99] and [Mac03], Hopfield Neural Networks are used to solve general combinatorial optimisation problems. Also in [Hei94] it is mentioned, that they are actually capable to solve mapping problems. So the first choice to solve the static mapping problem was a Hopfield Neural Network.

Unfortunately it turned out during the implementation and testing, that this class of algorithm is quite complex. The Hopfield Neural Networks posses a lot of degrees of freedom and parameters. It has to be decided, if the output of neurons should be continuous or discrete, and if it should be deterministic or stochastic. Additionally a suiting activation function has to be chosen. After that, the cost function of the given problem has to be integrated into the network and the biases of the neurons. Often, as in the case of the static mapping, a constraint term has to be defined and added to the cost function. Finally, an algorithm has to be chosen and to run the network to its equilibrium were the solution is achieved.

Many decisions are necessary to come to a result. In the implementation the network, described in chapter 4.3.3, found valid solutions. That means, the constraint, that every task has to be mapped to exactly one processor, was not broken. But the resulting mappings were not quite good. One problem was, that the mappings tended to put all tasks on one processor, even though the cost function had its minima on other locations.

Retrospective it can be said, that the realised Hopfield Neural Networks found sometimes very good mappings. But most of the time it found very poor solutions. To get good results much more exploration of the parameters and degrees of freedom is needed.

The second approach to solve the static mapping was Simulated Annealing. This algorithm is pretty easy. There is no need for any additional constraint, because it just checks valid possibilities. The only consideration one has to do, is which has to be chosen is a cooling schedule. This means, a temperature is needed from where the algorithm starts, a temperature where it terminates and a schema to vary the temperature. The starting and finishing temperature can be simply get out by trying some values. With the finishing temperature the probability should be near zero to transition into any other worsen state. And with the starting temperature the probability should be near 1 for a transition, even into very bad states.

The suitable cooling schedule, discussed in chapter 4.3.4, was determined, again by doing some experiments. During this experiments, it also turned

out, that it is nevertheless possible, that the algorithm gets stuck in local minima of the cost function. To avoid this, several passes are done from different starting points. Some starting points were taken by random and others by putting all tasks onto one processor. This method showed up quite good results. It was even though much faster as the usage of Hopfield Neural Networks which even resulted in worth results.

5.3 Results of Simulations

To verify that the dynamic remapping improves the mappings of the static phase in cases were its assumptions are broken, the same tests were done as discussed in chapter 4.3.5. The results are shown in figure 5.3. It can be seen, that the mappings, which are enhanced with dynamic load balancing are much better then without. They tend to be as good as if all tasks were mapped.

A test was invented, which should evaluate the quality of the combination of the chosen methods. In this test the simulator will simply run several different programs with overlapping execution times. The results are depicted in figure 5.4. Unfortunately the shown measures do not show a sharp result as it was expected. One have to look precisely at the figure to see that in cases where unmapped tasks appeared, the mappings with dynamic remapping are most often better then the mappings without.

The seemingly missing data points in this figure are only in the “fake” series. These missing points correspond to tests in which all tasks were mapped at the static phase. So there was no unmapped task at the execution time.

Another aspect is seen in figure 5.5. This figure shows data of the same test simulations. One can see, that in nearly all cases the taking of equal static phase weights results in worse runtimes. In contrast to this, the mappings with weights, which were generated after the calculus, discussed in chapter 5.1 results in shorter runtimes. This is shown for both cases, when dynamic remapping was done, and when it was not done.

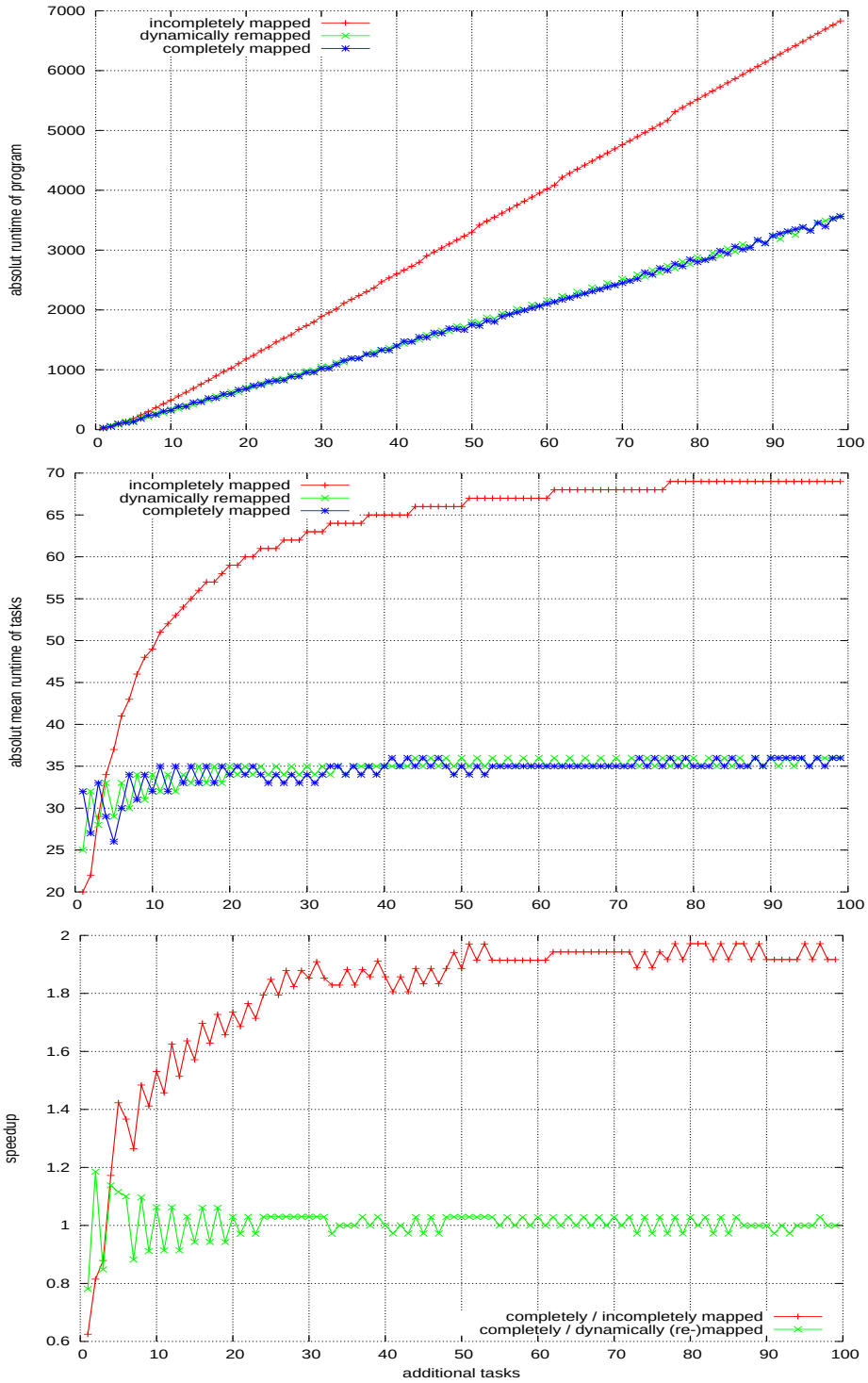


Figure 5.3: results of the tests with the graphs shown in figure 4.11; above: absolute runtimes of programs; middle: mean runtime of tasks, see definition on page 16; below: $\frac{\text{incompletelyMappedRuntime}}{\text{completelyMappedRuntime}}$ resp. $\frac{\text{dynamicallyRemappedRuntime}}{\text{completelyMappedRuntime}}$ = the ratio between the runtimes, when all tasks were mapped and some were not resp. dynamically remapped; additional tasks on the x-axis means the amount of tasks which arise at execution which were unknown to the mapping unit

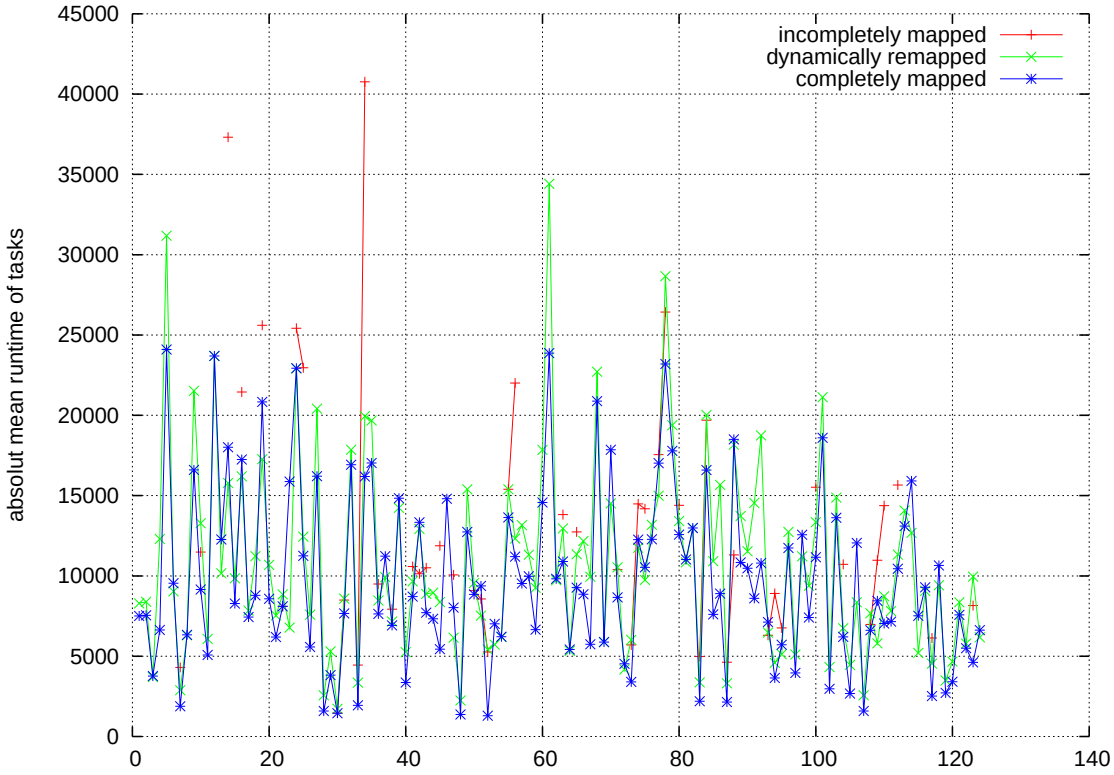


Figure 5.4: several randomly chosen programs running with overlapping execution times

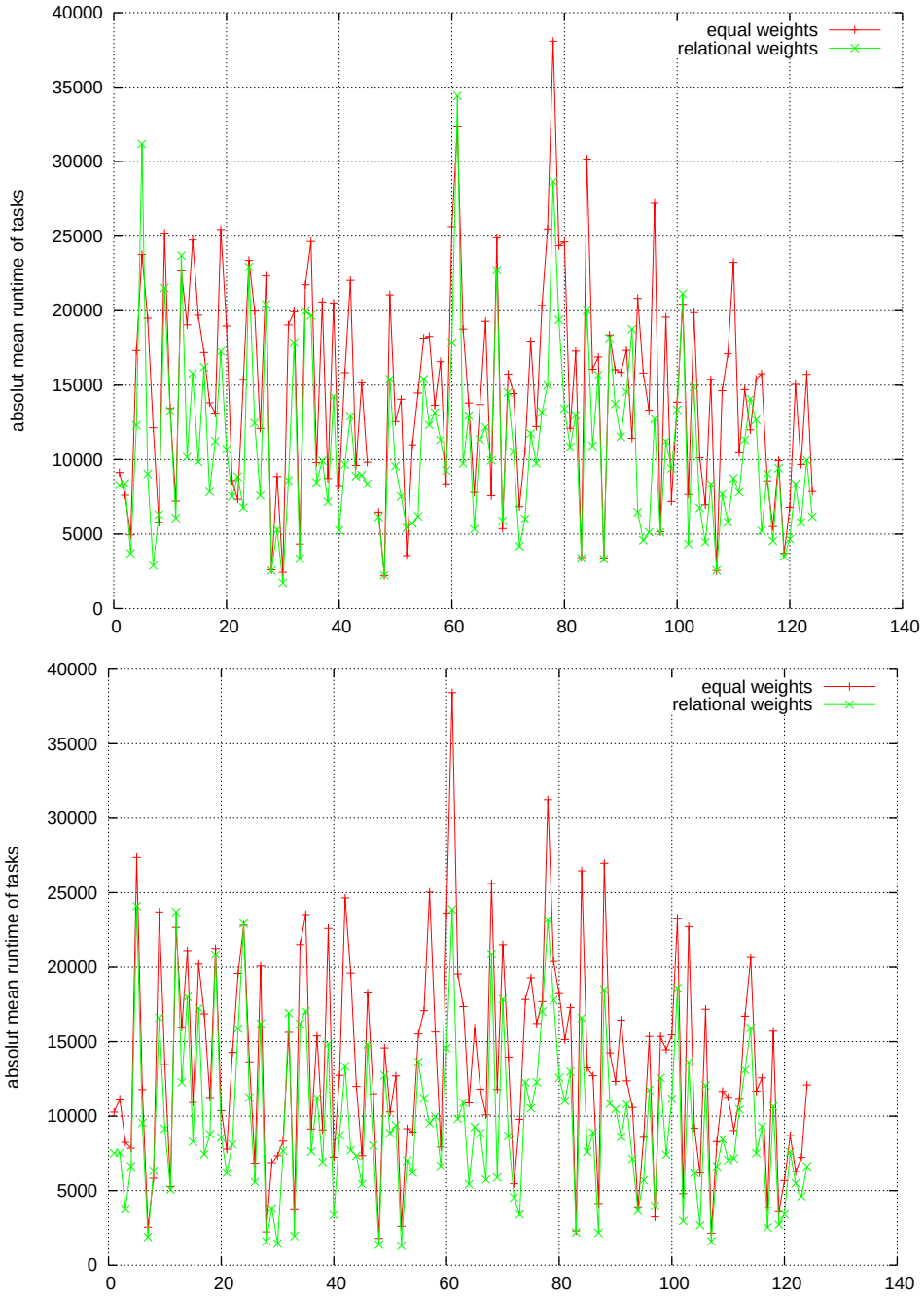


Figure 5.5: several randomly chosen programs running with overlapping execution times; above: with dynamic remapping; below: without dynamic remapping; *equal weights* denotes the mappings with equal static phase weights, *relational weights* denotes the mappings where the weights were calculated after the calculus which is discussed in chapter 5.1

5.4 Evaluation

As a result of this work it can be said, that the combined approach bears the potential to overcome the demands on such a system:

- low overhead
- fast response times of programs

But with the current realisation, these demands are met only in some cases.

6 Outlook and further work

As the whole work implies, especially the last chapter 5, the approach taken here is far away from being ready for the application in productive environments. This work just shows that this approach may be good in general, but currently it performs well only in some special cases. It shows, that, to improve the performance, there are still many accommodations to be done in nearly all parts of this method. The following subsections deal with the aspects, which arose during the implementation.

6.1 Static Mapping

In chapter 4.3.2 the simple exhaustive search algorithm was discussed, which was realized in this work. It bases on the quality measure, which is defined by the pseudo code in figure 4.6. This measure was defined to determine the degree in which the shape of the simulated runtimes of a program, plotted over all mappings, coincide with the corresponding values of the weight dependent cost function. So a mapping found at a global minimum of the cost function should correspond to a global minimum in the runtimes. With this mapping, the runtime should be minimal. The used measure, to determine the quality, has the disadvantage to be uninterpretable. A better choice would be to take the linear correlation function [PTVF92]. This function matches the requirements and its resulting values are easily interpretable. Its computational complexity is similar to that of the used quality measure. Some tests demonstrated, that this measure results in equal good weights as the ones, generated during the usage of the used quality measure. The correlation was discarded because in some cases the used search algorithm did not converge respective got stuck in an infinite loop.

Another possibility as quality measure would be to use a simulator instance with Simulated Annealing as static phase mapping algorithm. The resulting runtimes could be used as quality measure. This would greatly speed-up the calculus because the runtimes of all mappings, which are needed by the quality measures above, would no longer be needed. With this method,

the search algorithm simply searches directly for good weights. It does not try to find weights which form the cost function as the simulated runtimes. If this is really faster and if the resulting weights are really good[1] has to be tested in additional studies.

The simple search algorithm was chosen, because the effort of finding good weights was inestimable a priori. It is very likely, that this task can be done in a better way with respect to the speed of convergence and the quality of the results. A candidate would be Simulated Annealing, which, in contrast to the already discussed variant in chapter 4.3.4, is also capable to solve continuous optimization problems. The implementation of the GNU Scientific Library [GSL] is well adjusted to this need, and further methods can be found in [SBDH97], [CMMR87] and [GDM92].

The two already presented aspects concerning the cost function of the static phase mapping, chapter 4.3.1, which should be investigated in further work are:

- the modifications of the instruction term
- the fusion of the terms for communications and dependences

With the modifications of the instruction term the resulting behavior of the whole static mapping can be optimized. But the prize for this is an increased complexity because of the additional parameters which have to be set.

The fusion of the terms for communications and dependences will decrease the complexity of the static mapping problem. The disadvantage is the loosing of control over the single weighting of the terms. Interesting for further studies would be to delve the ratio of the advantage to the disadvantage.

Finlay, Simulated Annealing as such is a centralized algorithm within its commands are executed in sequence. But there are parallelized variants available, e.g. [MKBW92], so that the algorithm can run distributed on several nodes of a cluster system. This would be interesting to explore especially for the final running of the whole framework on real cluster systems consisting of many nodes.

6.2 Dynamic Load Balancing

The potential function of the dynamic remapping phase consists, like the static phase cost function, of weighted terms. In this work, these weights were simply guessed values which passed some tests and finally turned out

[1] the possibility of suboptimal solutions has to be considered

to be quite good. So they are constant values. At this point further research could find better weights due to strategic searches. Probably an adaptive rule can be found which generates the weights out of the parameters of the soft- and hardware, similar to the procedure for the static phase weights.

A possibly additional enhancement to the potential function, which should be investigated in further work, is the distinction of the tasks to already running and not yet running tasks. This can be usefully because a not yet started task could be dynamically mapped to another processor. So the influence of not running tasks should be smaller to reduce the risk of bad remapping decisions based on such tasks. Optimally, all terms would be additionally weighted for every task with the probability that the given tasks does not migrate. Because this probability is hardly to get, it would be a very broad approximation to weight the terms of not yet running tasks with a lower value.

Similar to the cost function of the static phase, the modifications of the instruction term should be examined, likewise the fusion of the terms for dependences and communications. The same arguments are valid for the potential function of the dynamic phase.

6.3 Framework

So, taking all together, there is a lot to do, to bring the discussed approach to its optimal performance. When this is done, this part have to be integrated into the mentioned framework, which consist of the prediction unit of [Gra04] and the profiling unit of [Sch04]. The presented mapping unit including the load balancing is linked only in a thematically manner to both other units. Also bases the implementation currently on a simulator and does not map any real task. So, finally the simulator must be decoupled and a real mapping enabling interface has to be integrated.

After that all is done, then it will be seen, if the framework performs well. Maybe the units have to be further integrated to optimize their communication and collaboration.

A Deutsche Zusammenfassung - German Abstract

In der vorliegenden Diplomarbeit wird ein Verfahren vorgestellt und untersucht mit dem parallele Programme automatisiert und effizient auf Cluster Computern ausgeführt werden sollen. Es handelt sich bei dem Verfahren um eine Kombination aus statischen Mapping beim Programmstart und dynamischem Lastausgleich während der Programmlaufzeit. Programme werden als Menge von Tasks gegesehen, elementaren Einheiten, welche mittels bewerteter Kommunikationskanten und Abhängigkeitskanten zu einem *Software Graphen* zusammen gefasst sind. Ebenso fließt die Hardwarebschreibung als Graph in das Verfahren ein. In *Hardware Graphen* entsprechen die Knoten den Prozessoren des Cluster Computers und die Kanten den Kommunikationsverbindungen.

Im untersuchten Verfahren wird nun zu jedem Programmstart, *statische Phase* genannt, jede Task des Programms einem Prozessor zugewiesen. Diese Mapping-Entscheidungen basieren auf der aktuellen Lastsituation und der Strucktur des Software-Graphen. Dafür wird eine Kostenfunktion aufgestellt deren Minimierung zu einem guten Mapping führen soll. Da diese Problemstellung auf ein kombinatorisches Optimierungsproblem hinausläuft, existiert eine Vielfallt an Algorithmen für diese Aufgabe. In dieser Arbeit wurden *Hopfield Neuronale Netzte*, sowie *Simulated Annealing* untersucht.

Während der *dynamischen Phase*, der Ausführungszeit eines Programms, wird zum Start jeder Task untersucht, ob die Task nicht doch besser auf einem anderen Prozessor starten soll als dem, dem sie durch die statische Phase zugeordnet wurde. Dies kann in folgenden Situationen der Fall sein:

- gestiegene Last auf dem Zielprozessor
- gesunkene Last auf Nachbarprozessor
- startende Task ist unbekannt, also noch nicht gemappt

- Fehlen einer ursprünglich gemappten, also eingeplanten Task auf Nachbarprozessor

Falls sich eine dieser Situationen ergibt, wird das Mappen zu den verschiedenen Prozessorkandidaten verglichen und die beste Variante ausgewählt. Dies geschieht nach dem Physikalischen Modell der Kräfte beziehungsweise Potentiale. Dazu wird ähnlich der Kostenfunktion in der statischen Phase eine Potentialfunktion aufgestellt, deren Minimierung zu einem guten Mapping führen soll. Diese wird dann für alle Prozessorkandidaten ausgewertet.

Beide Funktionen haben sich als sehr ähnlich herausgestellt. Beide bestehen aus einzelnen gewichteten Termen, die die jeweiligen Teilziele repräsentieren. Die Teilziele sind unter anderem die Minimierung des Kommunikations- und Rechenkosten. Das Finden von geeigneten Gewichten hat sich für die Potentialfunktion der dynamischen Phase als einfach erwiesen. Sie wurden durch Probieren und einigen Tests zu konstanten Werten gesetzt. Die Gewichte der Kostenfunktion der statischen Phase haben sich als sehr schwer bestimmbar heraus gestellt. Kleine Änderungen an ihnen können erhebliche Auswirkungen auf die Qualität der resultierenden Mappings haben und konstante Werte scheinen auch nicht für verschiedene Programme passend zu sein.

Somit hat die Suche nach einer Regel zur Bestimmung der Kostenfunktionsgewichte einen großen Anteil an der gesammten Arbeit ausgemacht. Es konnte ein einfacher grober Zusammenhang gefunden, mit dem aus den Eigenschaften der Software, wie beispielsweise die Summe der Instruktionen aller Tasks, die Gewichte bestimmbar wurden. Dieser Zusammenhang hat in einigen Versuchen nutzbare Ergebnisse erbracht. Leider hält dieser Ansatz der Gewichtsbestimmung aber nicht allen Tests stand.

Bibliography

[CMMR87] A. Corana, M. Marchesi, C. Martini, S. Ridella: *Minimizing multimodal functions of continuous variables with the "simulated annealing" algorithm*, ACM Transactions on Mathematical Software (TOMS), Volume 13 Issue 3, September 1987

[Doxy] URL of Doxygen: http://www.doxygen.org (visited: December 29, 2004)

[Ere02] N. Eremic: *Graph-Matching mit Hopfield-Netzten - Vergleich ausgewählter Verfahren*, diploma thesis at Institute for Artifical Inteligence at Technical University Berlin, December 2002

[FJMPJ] H. Franke, J. Jann, J.Moreira, P. Pattnaik, M. Jette: *An Evaluation of Parallel Job Scheduling for ASCI Blue-Pacific*

[Fos95] I. Foster: *Designing and Building Parallel Programs*, Addison-Wesley, 1995, URL http://www.mcs.anl.gov/dbpp/ (visited: December 29, 2004)

[FRS97] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, P. Wong: *Job Scheduling Strategies for Parallel Processing*, Springer Verlag, 1997, URL http://citeseer.ist.psu.edu/feitelson94theory.html (visited: December 29, 2004)

[GDM92] S. B. Gelfand, P. C. Doerschuk, M. Nahhas-Mohandes: *Theory and application of annealing algorithms for continuous optimization*, Proceedings of the 24th conference on Winter simulation, December 1992

[Gra04] M. Grahl: *On- und Offlineanalyse von verteilten Programmen*, diploma thesis at Institute for Operating- and Communication systems at Technical University Berlin, April 2004

[GSL] URL of the GNU Scientific Library (GSL): http://www.gnu.org/software/gsl/ (visited: December 29, 2004)

[Hay99] S. Haykin: *Neural Networks - A Comprehensive Foundation*, 2^{nd} edition, Prentice Hall, 1999

[Hei94] H.-U. Heiß: *Prozessorzuteilung in Parallelrechnern*, BI-Wissenschaftsverlag, Reihe Informatik, Volume 98, Mannheim, 1994

[HKP91] J. Hertz, A. Krogh, R. G. Palmer: *Introduction to the theory of neural computation*, ABP Perseus Books, 1991

[Hop82a] J. J. Hopfield: *Neural networks and physical systems with emergent colective computational abilities*, Proceedings of National Academy of Sciences, USA, 79, 1982

[Hop82b] J. J. Hopfield: *Neurons with graded responses have collective computational properties like those of thw-state neurons*, Proceedings of National Academy of Sciences, USA, 81, 1982

[Jag95] A. Jagota: *Approximating maximum clique with a Hopfield network*, IEEE Trans. Neural Networks, 6:724-735, 1995

[KGV83] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi: *Optimization by simulated annealing*, Science, 220: 671-680, 1983

[LA87] P. J. M. Laarhoven, E.H.L. Aarts: *simulated Annealing: Theory and Applications*, D. Reidel Publishing Company, Dordrecht, Holland, 1987

[Mac03] D. J. C. MacKay: *Information Theory, Inference, and Learning Algorithms*, 7^{th} version, Cambridge Univerity Press, 2003, URL http://www.inference.phy.cam.ac.uk/mackay/itila/ (visited: December 29, 2004)

[MCP43] W. S. McCuloch, W. Pitts: *A Logical Calculus of Ideas Immanent in Nervous Activity*, Bulletin of Mathematical Biophysics 5, 115-133, 1943

[MKBW92] P. P. Mutalik, L. R. Knight, J. L. Blanton, R. L. Wainwright: *Solving combinatorial optimization problems using parallel simulated annealing and parallel genetic algorithms*, Proceedings of the 1992 ACM/SIGAPP symposium on Applied computing: technological challenges of the 1990's, March 1992

[MPI] MPI URL http://www.lam-mpi.org/tutorials/nd/ (visited: December 29, 2004)

[PDJM04] A. Pillekeit, F. Derakhshan, E. Jugl, A. Mitschele-Thiel: *Load-based Intersystem Handover*, Shaker Verlag, Conference Proceedings, Volume 2, Technische Universität Ilmenau, 2004

[PTVF92] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery: *Numerical Recipes in C - The Art of Scientific Computing*, 2^{nd} edition, Cambridge Univerity Press, 1988-1992, URL http://www.nr.com (visited: December 29, 2004)

[RR04] T. Rauber, G. Rünger: *Parallele und verteilte Programmierung*, Springer-Verlag 2000

[SBDH97] P. Siarry, G. Berthiau, F. Durdin, J. Haussy: *Enhanced simulated annealing for globally minimizing functions of many-continuous variables*, ACM Transactions on Mathematical Software (TOMS), Volume 23 Issue 2, June 1997

[Sch04] J. Schneider: *Ablaufvorhersage für verteilte Programme mit Hilfe von Graphtransformation*, diploma thesis at Institute for Operating- and Communication systems at Technical University Berlin, April 2004

[Sch91] M. Schmitz: *Entwurf und Analyse eines verteilten Algorithmus zur Lastverteilung in parallelen Systemen*, diploma thesis at Institute for Operating- and Dialoguesystems at University Karlsruhe, December 1991

[Smi99] K. A. Smith: *Neural Networks for Combinatorial Optimization: A Review of More Than a Decade of Research*, INFORMS Journal on Computing, Volume 11, No. 1 Winter 99

[Xerces] URL of Xerces: http://xml.apache.org/ (visited: December 29, 2004)