Designing Parallel Programs of Parameterized Granularity

PROEFSCHRIFT

TER VERKRIJGING VAN DE GRAAD VAN DOCTOR AAN DE TECHNISCHE UNIVERSITEIT EINDHOVEN, OP GEZAG VAN DE RECTOR MAGNIFICUS, PROF. DR. J.H. VAN LINT, VOOR EEN COMMISSIE AANGEWZEEN DOOR HET COLLEGE VAN DEKANEN IN HET OPENDAAR TE VERDENGEN OP DINSDAG 10 NOVEMBER 1992 OM 16.00 UUR

DOOR

Pieter Struijk

GEBOREN TE ROTTERDAM
Dit proefschrift is goedgekeurd door de promotor

prof. dr. M. Rem

en de copromotor

dr. A. Kaldewaij
## Contents

1 Introduction .................................................. 1

2 Fine-grained Parallelism .................................... 5
   2.1 Basic Concepts and Notational Conventions .............. 6
   2.2 Specifications .............................................. 11
   2.3 Input/output Relations ..................................... 13
   2.4 Set of Equations ............................................ 17
   2.5 Communication Behavior .................................. 25
   2.6 Performance .................................................. 29

3 Granularity .................................................... 35
   3.1 Multiple Processes per Processor ......................... 36
   3.2 Communication Overhead .................................... 38
      3.2.1 A Technique for Enlarging the Grain Size of Programs .... 38
      3.2.2 Complexity Analysis for the $[M,K]$-transformation ......... 39

4 Deriving Programs of Parameterized Granularity .......... 42
   4.1 Pipe-line Computations ..................................... 42
   4.2 Stack-like Computations .................................... 46

5 General Scheme ............................................... 52
   5.1 Pipe-line Computations ..................................... 52
   5.2 Stack-like Computations .................................... 56

6 Experimental Results ........................................ 61
   6.1 Processor Networks ......................................... 61
      6.1.1 A Single Process per Processor ...................... 62
      6.1.2 Multiple Processes per Processor ..................... 63
      6.1.3 Implementation Details ................................ 67
   6.2 VLSI ............................................................ 68
      6.2.1 Simulation Results ...................................... 68
      6.2.2 Area ....................................................... 69
      6.2.3 Cycle Time ................................................ 69

7 Concluding Remarks .......................................... 72

Bibliography .................................................... 76
Contents

A General Scheme for Stack-like Computations
A.1 Statement $S'$ ........................................ 79
A.2 Statement $S''$ ....................................... 81
A.3 Statement $S'''$ ...................................... 82
A.4 An Optimization ..................................... 84

Index .................................................................. 88

Samenvatting .................................................. 89
Acknowledgements

First of all, I would like to thank Martin Rem for offering me the possibility of doing research in the challenging area of parallel processing at Eindhoven University of Technology.

The work in this thesis has been supported by Philips Research Laboratories Eindhoven. I thank Eddy Odijk for this and for the fact that I was allowed to visit Philips Research once every week. I enjoyed my visits a lot.

Martin Rem and Anne Kaldewaij are gratefully acknowledged for their careful reading of draft versions of this thesis and valuable suggestions for improving its readability. I thank Johan Lukkien for discussing Chapter 6 with me.

Finally, I would like to thank my colleagues with whom I shared my office: Wim Kloosterhuis, Berry Schoenmakers, John Koster, Ad Peeters, Joost Katoen, and, at Philips, Frans Sijstermans. It was fun!
Chapter 1

Introduction

The subject of this thesis is the design of parallel programs. Parallel programs are programs that are executed by a parallel machine, i.e. a computer equipped with a number of processors. The type of parallel machine that is considered in this thesis is one that has many identical processors, each executing its own code and interacting with each other by exchanging messages along (hardwired) links. Furthermore, each processor has access to its own local memory. Such parallel machines have become popular over the last few years, since they offer a cost-effective way of increasing computational power.

Much effort has been put on the design of fast processors and fast communication hardware. Relatively less attention has, however, been paid to the question of how these machines can be programmed effectively. Since increasing execution speed is the main reason for doing parallel computing, it is important that parallel programs are specifically designed to exploit the possibilities of parallelism as much as possible.

The availability of increased computational power can be exploited in a number of ways. One way exploits, of course, the fact that the outcome of a computation is available in a shorter time. Take, for instance, an operator of a chemical plant. The operator's job is to control the plant in such a way that raw materials are transformed into products in the most efficient way, i.e. optimal yield of the desired products and minimal yield of useless by-products while using minimal energy and minimal raw material. In these kind of control problems usually a large number of parameters are involved. If the required quantities of products and the quantity of raw material change on a daily basis, the algorithm to compute a control should obviously not take half a day, but at most 15 minutes, say. If, in addition, the operator wants to study the implications of small adjustments of some parameters, the execution time of the algorithm should yet be smaller. Another way of taking advantage of increased computational power is that in the same time in which an algorithm solved a problem beforehand, now a larger problem can be solved. In the operator's example, increased computational power would allow to compute a control that also takes into account the quality and composition of raw materials and takes into account expulsion of products that pollute the environment. As another example, in the area of signal processing it is often required that the output signal has the same frequency as the input signal. A decoder for a CD-player has to meet real-time constraints to guarantee that signals can be transformed into music of a high quality.
Writing a program that satisfies its specification is a difficult task. Basically, there are two approaches. In the first approach, programs are written and then it is verified whether they meet their specification. A second approach is to design programs from their specification by applying correctness-preserving transformation-rules such that the resulting program is correct by design. The latter approach is attractive for a number of reasons. First of all, proving the correctness of a program, as a whole, is a difficult and elaborate task (even impossible for large parallel programs). Secondly, correctness by design offers the possibility of designing programs in a calculational style. If at a certain step in the design a number of possibilities for the next step emerge, the choice for one particular design decision can often be guided by performance considerations. Furthermore, in a calculational design style one can often benefit from knowledge gained from similar problems or subproblems.

In the field of design techniques for sequential programs much research has been done [4, 6]. Less attention has been paid to techniques for designing parallel programs. The research effort in this topic has, however, increased due to the availability of massively parallel machines these days. Interest in parallel computing is mainly restricted to the scientific world. One of the reasons is that writing parallel programs is much more difficult than writing sequential programs. The non-deterministic behavior of most parallel programs (a collection of co-operating sequential programs) makes it almost impossible to verify that a program meets its specification. Therefore, solid design methods for constructing parallel programs are indispensable.

In the literature, methodologies for parallel programming are often distinguished as —but not restricted to— processor farming, algorithmic parallelism, or geometric parallelism [21]. Processor farming can typically be applied to problems that consist of a (large) number of largely independent tasks. A controller distributes the tasks among the processors in the parallel machine evenly and collects the results. If processor farming is applicable, it should certainly be considered, since it usually offers good results. Algorithmic parallelism can be applied if the (sequential) algorithm consists of a number of stages that can be distributed such that each processor executes a different stage. Pipelines, in which each processor receives a datum, processes it, and passes it to the next processor, are a typical example of algorithmic parallelism. Geometric or data parallelism can be applied if the data that has to be processed can be distributed over the processors, such that at regular time intervals exchange of data between neighboring processors takes place. One could say that in algorithmic parallelism the code is distributed over the processors, whereas in geometric parallelism the data are distributed over the processors.

Much research has been done on extracting parallelism from sequential programs (cf. [10], e.g.). Research efforts in this area are mainly triggered by the fact that large existing libraries of sequential code had to be ported to parallel machines without redesigning code. It has turned out that parallelizing sequential code for execution on a number of processors is hard and often does not yield good results.

This thesis advocates the following approach for designing parallel programs, which follows the opposite direction from the one above:

1. From a specification we design a fine-grained parallel program, i.e. a large collection of small sequential programs (called processes) from which each exhibits
a simple behavior, e.g. repeatedly exchanging data with neighbor processes and computing a simple function on received data.

2. Transform the fine-grained program into a coarse-grained program.

With respect to the first step, it is shown for a large class of specifications how such fine-grained programs can be derived. We restrict ourselves to so-called data-independent programs, i.e. the program's functionality and its interaction with neighbor processes, its so-called communication behavior, can be considered independent of each other. The ratio of the number of communications and the number of computations of a fine-grained program is high. Therefore, fine-grained programs can be implemented most effectively on a medium that supports fast communication, such as VLSI circuits.

When fine-grained programs are implemented on a parallel machine, however, communication is much slower and tends to be the bottleneck of the system. Therefore, the second step above has to be applied. Reducing the time spent on communicating yields a coarse-grained program which is more suitable for execution on a processor network. (As a simple example of reducing communication time, assume a process that sends the result of expression $E$ to another process along channel $c$. The sending process executes output statement $c!E$ and the receiving process executes input statement $c?z$. If both processes are merged into a single process, communication along channel $c$ can be replaced by the assignment $z := E$.)

The derivation of a parallel program, satisfying some specification, involves a decomposition of the overall problem into a collection of processes with interconnecting channels between them. From the overall specification it is deduced what values are communicated along the channels. From these specifications of the channels one derives how an output value of a process is computed from input values and local variables. This is called the set of equations that the computation of output values should satisfy. Finally, from this set of equations the behavior of each process is derived. The design of a the set of equations is a task in which the programmer's creativity often plays an important role.

The literature reports on much research in the field of synthesizing VLSI processor arrays (which can be considered to be fine-grained parallel programs) [3, 12, 19, 22, 23, 24]. These methods aim at some, more or less, automated design of programs: specifications are compiled into processor arrays. The input specifications of such a compiler can be compared with the set of equations of our method and is usually a set of recurrence equations of some special form. If the (initial) set of recurrence equations is not feasible, it is, either manually or automatically, transformed into a feasible set of recurrence equations. Automatic synthesis usually does not address how the set of recurrence equations is obtained from the overall specification of the problem to be solved.

The transformation of a fine-grained parallel program into an efficient coarse-grained parallel program depends on the implementation medium. In particular, the way communication is implemented is important, since communication is overhead of a parallel program with respect to its sequential counterpart. In this thesis, both implementations as VLSI circuits and implementations on processor networks are discussed. To cope with the different characteristics of these implementations, fine-grained programs are transformed into parameterized ones. By choosing appropriate values for the
parameters, a parameterized parallel program can be tailored to the specific properties of the implementation medium.

Overview

In Chapter 2 a method for designing fine-grained programs is discussed. First, basic concepts and notational conventions are presented by means of a simple example. In subsequent sections the design method is explained by identifying five phases in the design of a parallel program. The fact that each phase is addressed separately does not imply that they are completely independent of each other. A design decision made in one phase usually has impact on some next design phases. In this chapter, two programs that serve as running examples throughout this thesis are derived from their specifications.

Chapter 3 considers the implementation of parallel programs on processor networks. A simple model is presented so as to analyse an implementation's speedup, which is a measure for how much faster the parallel program is than its sequential counterpart. It turns out that the overhead introduced by communication statements should be reduced to allow efficient execution. A technique for transforming fine-grained programs into programs of a coarser grain is discussed. This technique yields programs of parameterized granularity.

Chapter 4 applies the proposed technique to two types of programs. As specific instances, the running examples are transformed into parallel programs of parameterized grain size.

Chapter 5 shows that the transformation technique is generally applicable to any fine-grained program.

In Chapter 6 experimental results for both VLSI and processor network implementations of the running examples are presented.

Finally, some concluding remarks are presented in Chapter 7.
Chapter 2

Fine-grained Parallelism

In this chapter we present a method for designing parallel programs. A parallel program represents a network of processes that interact with each other by exchanging messages along channels. Channels can be either internal, i.e. between two processes, or external, i.e. between a process and the environment of the program. Along external channels input can be sent to the network or output can be retrieved from the network.

A specification expresses the relation between input and output of a program. Such a relation is called an input/output relation, i/o-relation for short. The way programs are specified is discussed in Section 2.2. The computations that we consider are 'stream oriented' and the individual elements of an output stream are defined in terms of the input streams. With each input or output stream an external channel is associated.

In Section 2.3 it is shown how a program specification can be decomposed into a collection of processes. This decomposition proceeds in a calculational way and results in a collection of i/o-relations for both internal and external channels. Such an i/o-relation of a channel specifies the values that are communicated along that channel. In this phase of the design, the topology of the process network is also determined.

The resulting i/o-relations are specifications of individual processes. In Section 2.4 we determine how a process can compute values sent along its output channels in terms of values received along its input channels. In this design phase, it is possible that additional (internal) channels are introduced, resulting in additional i/o-relations. It is also possible that local variables are introduced. At the end, we obtain a set of defining equations which expresses how values to be communicated along output channels are computed.

Such a set of defining equations induces an order on the communication actions of a process. If an input value contributes to an output value of a process, then that process should receive that input value before it can produce the output. In Section 2.5, so-called communication behaviors for the processes of a program are determined. These communication behaviors are constructed in such a way that the resulting parallel program is deadlock-free. Given a set of defining equations and a communication behavior consistent with that, the program text of each process can be given.

Finally, in Section 2.6, we address some concepts related to the performance of parallel programs. In particular, we consider two ways of implementing parallel pro-
grams, viz. an implementation on a processor network and an implementation as a VLSI circuit. When implementing a program on a processor network we are primarily interested in how fast the program can compute its output. When implementing a program as a VLSI circuit we are also interested in the data size of the program, since this determines the chip area needed which is one of the main concerns in circuit design.

In this way, five design phases can be identified. Each section of this chapter addresses one phase. This does by no means imply that these phases are independent of each other. Design decisions made in one phase usually have a great impact on some next design phase. For instance, different choices for the i/o-relation of a channel may result in programs of different performance.

2.1 Basic Concepts and Notational Conventions

In this section we illustrate the basic concepts and notational conventions that will be used in the sections that follow. We refer to some concepts in a rather informal way. The theoretical basis, together with theorems that we apply, has been dealt with extensively in [25, 32].

Program notation

Consider a parallel program that consists of a collection of N processes. Processes are numbered from 0 up to N–1. The program text of process k (k≠N–1) reads

\[
[a_k?va \\
; vb := va \\
; (b_k?vb,a_{k+1}?va \\
; a_k?va,b_{k+1}?vb \\
; vb := va + vb \\
)^
\]

and the program text of process N–1 reads

\[
[va : \text{int}; (a_{N-1}?va; b_{N-1}?va)^*]
\]

As can be observed, program texts are written in a CSP-like notation [7] with explicit communication between processes: for input channel c, c?x denotes the reception of a value communicated along channel c which is stored in variable x, and for output channel d, dE denotes the sending of the value of expression E along channel d. Communication is synchronous.

Each channel, except for external channels a_0 and b_0, is shared between two processes. For instance, channel a_{k+1} is an output channel of process k and an input channel for process k+1.

Let us have a closer look at the program text of process k (k≠N–1). In the first line, two variables va and vb of type integer are declared. Next, a statement list is
Input/output relations and defining equations

The $i$-th value ($i \geq 0$) communicated along channel $c$ is denoted by $c(i)$. From the program text we infer that process $k$ uses 2 input channels $a_k$ and $b_{k+1}$, and 2 output channels $b_k$ and $a_{k+1}$. The values communicated along output channels are computed from values communicated along input channels. This example satisfies the following set of equations

$$
\begin{align*}
a_{k+1}(i) &= a_k(i) \\
b_k(0) &= a_k(0) \\
b_k(i+1) &= a_k(i+1) + b_{k+1}(i)
\end{align*}
$$

for $i \geq 0$. From the program text of process $N-1$ we infer that $b_{N-1}(i) = a_{N-1}(i)$, for $i \geq 0$.

The program defines a network of processes (see Figure 2.1) that consists of a linear array of $N$ processes. The network has external input channel $a_0$ and external output channel $b_0$. The environment communicates with the program along these external channels. If the input of the program is input stream $A$, i.e., $a_0(i) = A(i)$ for $i \geq 0$, then the values communicated along the channels of the program satisfy the following input/output-relations

$$
\begin{align*}
a_k(i) &= A(i) \\
b_k(i) &= (\Sigma j : 0 \max (i+1-N+k) \leq j \leq i : A(j))
\end{align*}
$$

for $i \geq 0$. If the output stream produced by channel $b_0$ is denoted by $B$, the program computes the sum of the last $N$ received elements of the $A$ sequence:

$$B(i) = (\Sigma j : 0 \max (i+1-N) \leq j \leq i : A(j))$$

for $i \geq 0$. Hence, for $i \geq N-1$, we have

$$B(i) = (\Sigma j : i - N < j \leq i : A(j))$$
Communication behavior

Up to now we only discussed the functional behavior of the program. Other important issues in parallel programming are deadlock-freedom of a program and performance. With respect to these subjects we study the so-called communication behavior of the processes, i.e., the order in which a process accesses its channels. The communication behavior of process \(k\) (\(k \neq N-1\)) is

\[
CB_k : a_k; (b_k, a_{k+1}; a_k; b_{k+1})^*
\]

and the communication behavior of process \(N-1\) is

\[
CB_{N-1} : (a_{N-1}; b_{N-1})^*
\]

(Since channels are uni-directional, we omit '?' and '?' to denote channel types.) In the example, the communication behavior does not depend on the values that are communicated or computed. Therefore, the computation is called data independent. For data independent computations, the communication behavior and the functional behavior of the program can be considered separately.

A communication behavior induces an order on the communication actions of a process. We consider only a special kind of communication behaviors, viz., those expressed in terms of (choice-free) commands (called restricted commands in '92)). Commands are defined inductively as follows:

1. \(\epsilon\) is a command;
2. \(a\) is a command for channel name \(a\);
3. if \(S\) is a command not containing any star then \(S^*\) and \(S^n\) (\(n \geq 0\)) are commands;
4. if \(S\) and \(T\) are commands and \(S\) contains no star then \(S; T\) is a command;
5. if \(S\) and \(T\) are commands with disjoint sets of channel names then \(S; T\) is a command.

Here, '\(;\)' denotes sequential composition, '\(^*\)' and '\(^n\)' denote unbounded and bounded (\(n\) times) repetition, and '(', ')' denotes that no order is imposed on the execution of its operands.

Partial order on communication actions

A command induces a partial order on communications. We do not formally define the partial order induced by a command but present a few examples only.

Example 2.1 Partial order induced by a command

Let \('a_j < b_k'\) denote that the \(k\)-th communication along channel \(b\) occurs after the \(j\)-th communication along channel \(a\). For any channel \(a\) we have \(a_i < a_{i+1}\) for \(i \geq 0\).

Command \((a; b)^*\) induces \(a_i < b_j\) and \(b_i < a_{i+1}\).

Command \(a; (a; b)^*\) induces \(a_{i+1} < b_j\) and \(b_i < a_{i+2}\).
2.1. Basic Concepts and Notational Conventions

Command \(a;(b,a,c)^*\) induces \(a ti < b ti, b ti < a f(i+1), b ti < c ti,\) and \(c ti < b f(i+1).\) Notice that the \((i+1)\)-th communication along \(a\) and the \(i\)-th communication along \(c\) are not ordered.

Command \(a;(a;b,a,c)^*\) induces \(a f(2i+1) < b ti, b ti < a f(2i+2), b ti < c ti,\) and \(c ti < a f(2i+3).\)

\(\square\)

Not all partial orders can be expressed as a command. An example thereof is: \(a ti < b f(i+2)\) and \(b ti < a f(i+2).\) Command \((a,b)^*\) is the best we can find. However, this command induces \(a ti < b f(i+1)\) and \(b ti < a f(i+1),\) which is stricter than the required partial order.

Projection of commands

The order in which a subset of a command's channels are accessed can also be expressed by a command. Restricting a command to a subset of channels is called projection, denoted by \([\cdot]\). Consider communication behavior \(CB_k\) of process \(k.\) The order in which process \(k\) accesses channels \(a_{k+1}\) and \(b_{k+1}\) is

\[CB_k[\{a_{k+1}, b_{k+1}\}] = (a_{k+1}; b_{k+1})^*\]

Commands enjoy the property that projection on a set of channels can be performed by removing from the command all channels that are not in the set [32].

Deadlock

A program is deadlock-free if channels that are shared between processes are accessed in the same order. In the example, process \(k\) and its neighbor process \(k+1\) share channels \(a_{k+1}\) and \(b_{k+1}.\) To conclude deadlock-freedom, we require that \(CB_{k+1}[\{a_{k+1}, b_{k+1}\}] = CB_k[\{a_{k+1}, b_{k+1}\}].\) In our example we indeed have no deadlock since

\[CB_{k+1}[\{a_{k+1}, b_{k+1}\}] = (a_{k+1}; b_{k+1})^* = CB_k[\{a_{k+1}, b_{k+1}\}]\]

Minimal buffering

Consider the equation for output channel \(b_{N-1}\) of process \(N-1,\) viz. \(b_{N-1}(i) = a_{N-1}(i).\) The order imposed on these communication actions is that \(b_{N-1}(i)\) should occur after \(a_{N-1}(i)\) has occurred. Process \(N-1\) has communication behavior \(CB_{N-1} = (a_{N-1}; b_{N-1})^*\), which is more strict than necessary since in \(CB_{N-1}\) communications \(b_{N-1}(i)\) and \(a_{N-1}(i+1)\) are also ordered. Another communication behavior that is consistent with \(b_{N-1}(i) = a_{N-1}(i)\) is \(CB'_{N-1} = a_{N-1}; (a_{N-1}; b_{N-1})^*.\) In contrast to process \(N-1\), which requires only one variable \(v_2,\) a process with communication behavior \(CB'_{N-1}\) requires at least two variables: each time output \(b_{N-1}(i)\) is being sent, input \(a_{N-1}(i+1)\) has already been received and an additional variable is needed to store that value. We say that the communications along channel \(a_{N-1}\) are buffered. Communication behavior \(CB_{N-1}\) is preferred to \(CB'_{N-1}\) because fewer variables are needed. When constructing a communication behavior, we aim at a communication behavior with minimal buffering.
Sequence functions

In order to analyze the performance of a program we use so-called sequence functions [25, 32], which enables us to express when a communication along a channel or a computation within a process can be scheduled. A sequence function should respect the communication behavior of processes and it should, moreover, respect the duration of communications and computations.

Let \( \sigma \) be a sequence function. The value \( \sigma(c, i) \) denotes the time at which communication \( c(i) \) along channel \( c \) can be scheduled. To be more precise, an action takes place at time \( t \) means that that action is completed at time \( t \). Consider process \( k \) of the example and assume that a communication action takes time \( \alpha \) and that a computation \( (vb := va \text{ or } vb := va + vb) \) takes time \( \beta \). The time at which \( a_k(0) \) is completed takes at least \( \alpha \). Thus \( \sigma \) should satisfy

\[
\sigma(a_k, 0) \geq \alpha 
\]

Between \( b_k(0) \) and \( a_k(0) \) a computation is performed resulting in

\[
\sigma(b_k, 0) - \sigma(a_k, 0) \geq \alpha + \beta 
\]

No particular order is specified between the execution of communications \( b_k(i) \) and \( a_{k+1}(i) \). This is expressed as

\[
\sigma(b_k, i) = \sigma(a_{k+1}, i) 
\]

Other requirements for sequence function \( \sigma \) are

\[
\sigma(b_k, i) - \sigma(a_k, i) \geq \alpha + \beta \\
\sigma(a_k, i+1) - \sigma(b_k, i) \geq \alpha \\
\sigma(a_{k+1}, i) = \sigma(b_{k+1}, i) 
\]

Given those requirements, many solutions for \( \sigma \) are possible. One of these is

\[
\sigma(a_k, i) = \alpha(1 + 2i + 3i) + \beta(1 + 2i + 3i) \\
\sigma(b_k, i) = \alpha(2 + 2i + 3i) + \beta(1 + 2i + 3i) 
\]

Of all possible solutions for \( \sigma \), the above definition is the ‘strongest’ solution for the set of requirements, i.e. the inequalities of the requirements are replaced by equalities as much as possible (\( \sigma(a_k, 0) = \alpha \) and \( \sigma(b_k, i) - \sigma(a_k, i) = \alpha + \beta \)). This solution for the sequence function reflects that all processes execute at full capacity and that the environment—which is responsible for sending input values and receiving output values—is maximally co-operative.

Performance

From the sequence function of a program, we can infer performance related issues by looking at what time external input and output takes place.

The response time of a program is the time between successive external communication actions. Considering output channel \( b_0 \) of the example program, we have
2.2. Specifications

A parallel program represents a network of processes that interact with each other by exchanging messages along channels. Each channel is a uni-directional point-to-point connection between a sending and a receiving process. Broadcast, i.e. multiple receiving processes, is not allowed. Input and output channels provide a means for the environment to interact with a parallel program.

The specification of a parallel program consists of two parts. One part is the relation between the input and output values. Of course, it is not described in what way the output should be computed. In this respect a specification of a parallel program does not differ from a specification of a sequential program. The second part of the specification describes in which order the program should access its input and output channels. This is called the communication behavior of the program with respect to its environment.

We restrict ourselves to data-independent computations. For such computations the functionality and the communication behavior of a program can be considered separately. Often, the communication behavior of a program is not even specified. In that case, the communication behavior can be determined while designing the program. By postponing the decision what the communication behavior of the program should be, we introduce freedom for the designer. It is often possible to anticipate on the communication behavior a program will have. If we foresee that this communication behavior is not of a form we would like it to be, we adjust the specification. An example thereof will be presented in this section. Sometimes, the communication behavior of the program is specified in advance. We do so, in particular, when the program is part of a larger system and the interface between the program and the rest of the system has already been fixed.

The computations that we consider are ‘stream oriented’. The computation processes streams of input data and produces streams of output data. We give two examples, viz. a Palindrome Recognizer and the ‘Occurrence Count Last’ problem, that will serve as running examples throughout the rest of this thesis. These examples were chosen, since they represent a large class of computations.

Palindromes are words that are their own reverse, like ‘a’, ‘madam’, or ‘1991’.

Example 2.2 (Palindrome Recognizer)

Given an input stream \( A \) of symbols, compute output stream \( B \) satisfying

\[
B(i) = (\forall j: 0 \leq j \leq i : A(j) = A(i-j))
\]

for \( i \geq 0 \).
From this specification we infer that \( B(i) \) depends on the first \( i+1 \) elements of input stream \( A \). If we design a parallel program for recognizing palindromes that receives input stream \( A \) along input channel \( a \) and delivers output stream \( B \) along output channel \( b \), we expect the program to have communication behavior \( (a;b)^* \) —i.e., input channel \( a \) and output channel \( b \) are accessed in an alternating way—, since each \( B(i) \) can be computed after receiving \( A(i) \). Note that, with this communication behavior, each output is produced as soon as the last input value on which it depends has been received.

The second example of the type of computations we consider is a so-called window computation. A window is a consecutive segment of a stream. The 'Occurrence Count Last' problem, OCL for short, is a simple example of a window computation. In this problem, for each window of length \( N \) of the input stream it is determined how many times the last element of that window occurs in the window.

**Example 2.3 (Occurrence Count Last)**

Let \( N \geq 1 \). Given an input stream \( A \) of integers, compute output stream \( B \) satisfying

\[
B(i) = \left( \# j : i \leq j < i+N : A(j) = A(i+N-1) \right)
\]

for \( i \geq 0 \).

\( \square \)

Value \( B(i) \) is determined by window \( A[i..i+N] \). From this specification we infer that each successive element of stream \( B \) is obtained by computing a function on the previous window shifted over one position. As a result, a (parallel) program for OCL (or any other window computation) produces one element of the output stream for each element of the input stream it receives. There is, however, one exception: \( H(0) \), the first element of the output stream. This element depends on the first \( N \) elements of the input stream, since there is no previous window. From this observation we infer that a program will have communication behavior \( a^N;(b;a)^* \). A disadvantage of this communication behavior is its dependence on parameter \( N \). In the design of a parallel program, such a communication behavior often leads to a collection of processes that have different communication behaviors. For now, we take it as an advantage if the processes that constitute a computation have the same communication behavior. We, therefore, aim at specifications in which \( B(0) \) depends on \( A(0) \) only. This is achieved if we adjust the specification such that \( B(i) \) is a function of window \( A[i-N..i]\). As a consequence, we are only interested in \( B(i) \) for \( i \geq N-1 \). For \( i < N-1 \) any value for \( B(i) \) will do. It is, for instance, possible to treat \( B(i) \) for \( i < N-1 \) as a window computation involving (non-existing) negatively indexed elements of stream \( A \). As we will show later, it turns out that the design of the parallel program is not complicated if we choose appropriate values for these negatively indexed elements. We foresee three possibilities for specifying OCL when considering \( B(i) \) for \( i < N-1 \).

A first possibility is that \( B(i) \) is left unspecified for \( 0 \leq i < N-1 \). When designing the program, any appropriate value for \( B(i) \) will do. As a second possibility, let \( B(i) \) be specified as a computation on a window starting from \( A(0) \). For OCL we obtain

\[
B(i) = \left( \# j : i-N < j \leq i \land j \geq 0 : A(j) = A(i) \right)
\]
for \( i \geq 0 \). Finally, a third possibility is that \( B(i) \) is specified as a computation on window \( A(i-N..i] \) where the negatively indexed elements of stream \( A \) have appropriate values. For OCL we obtain
\[
B(i) = (\# j : i-N < j \leq i : A(j) = A(i))
\]
for \( i \geq 0 \).

For the OCL problem we adopt the third possibility:

**Example 2.4** (Occurrence Count Last)
Let \( N \geq 1 \). Given an input stream \( A \) of integers and appropriate values for \( A(j) \) with \( j < 0 \), compute output stream \( B \) satisfying
\[
B(i) = (\# j : i-N < j \leq i : A(j) = A(i))
\]
for \( i \geq 0 \).

The computations we focus our attention on are of a kind where each output depends on many input values and where each input value affects the value of many outputs. In Example 2.4, each output value depends on \( N \) input values and each input value is used for \( N \) output values. We do, in particular, not consider computations of which the (global) specification can be decomposed into a large number of independent tasks. Computations of this kind are easily parallelized by distributing these tasks over a number of processes, each computing a few tasks. Solutions of this kind are well-known and often referred to as processor farming [15, 17, 21].

### 2.3 Input/Output Relations

In a specification, the output of a program is specified as a function of the input. A parallel program receives input along input channels and produces output along output channels. In a specification only external input and external output channels are specified. A parallel program consists of a collection of cooperating processes sending messages along internal channels as well. As a consequence, when designing a parallel program there is a need for specifying the messages sent along internal channels. In this section we explain how to obtain the specifications, called input/output-relations, for the channels of the constituting processes of a parallel program.

The i/o-relations for the channels of the processes are generalized expressions of the original specification and are often obtained in a calculational way. In this approach there is at least one process that produces the specified output of the program. Furthermore, we see to it that the output of a cell can be produced by performing a computation on the data it receives from other processes. When generalizing the specification, we often have in mind a network topology of the parallel program. Topologies are, for instance, linear arrays (either bounded or unbounded), matrices, and tree structures.

In the following example, we generalize the specification of the Palindrome Recognizer of Example 2.2. We obtain an unbounded linear array of processes of which the first one can produce the specified output stream.
Example 2.5 (Palindrome Recognizer)
The specification of the Palindrome Recognizer of Example 2.2 reads: given an input stream $A$ of symbols, compute output stream $B$ satisfying

$$B(i) \equiv (\forall j: 0 \leq j \leq i : A(j) = A(i-j))$$

for $i \geq 0$.

We already observed that for the computation of $B(i)$ the values of $A[0..i]$ are needed. In the parallel program we would like $B(i)$ to be produced as soon as possible after receiving $A(i)$. When generalizing, we, therefore, consider the terms in the quantified expression that depend on $A(i)$: $j = 0$ and $j = i$. The symmetry between $j$ and $i-j$ is not reflected in the specification. We, therefore, rewrite the specification and obtain

$$B(i) \equiv (\forall j,h: 0 \leq j \land 0 \leq h \land j+h=i : A(j) = A(h))$$

for $i \geq 0$.

We, now, distinguish three cases: $B(0)$, $B(1)$, and $B(i+2)$. For $B(i+2)$ we derive,

$$B(i+2)$$

\begin{align*}
&\equiv \{ \text{specification} \} \\
&\equiv \{ i \geq 0: \text{split off } (j,h) = (i+2,0) \text{ and } (j,h) = (0,i+2) \} \\
&\quad \land (A(i+2) = A(0)) \land (A(0) = A(i+2)) \\
&\quad \land (\forall j,h: 1 \leq j \land 1 \leq h \land j+h = i+2 : A(j) = A(h)) \\
&\equiv \{ \text{shift range of quantification} \} \\
&\quad \land (A(0) = A(i+2)) \land (\forall j,h: 0 \leq j \land 0 \leq h \land j+h = i : A(j+1) = A(h+1))
\end{align*}

By observing the similarity between the specification and the quantified expression in the last line of the derivation, we conjecture a parallel program consisting of a linear arrangement of processes, numbered consecutively from 0 onwards (see Figure 2.2). Process $k$ has output channel $b_k$ of which the $i$-th communication satisfies

$$b_k(i) \equiv (\forall j,h: 0 \leq j \land 0 \leq h \land j+h = i : A(j+k) = A(h+k))$$

The first process, numbered 0, produces output stream $B$, since $B(i) = b_0(i)$. On account of the derivation above, we have $b_0(i+2) \equiv (A(0) = A(i+2)) \land b_1(i)$ for $i \geq 0$. \qed
In this example, we only considered output channels. We are in particular interested in producing each output value as soon as all input values on which it depends have been received, i.e. we are interested in programs of a low latency. In order to obtain a low latency, specifications are generalized by isolating those terms that contain input values that have been received last.

The above approach is applicable whenever output values depend on all preceding inputs. As a generalization, we consider an input stream $A$ and an output stream $B$ such that $B(i)$ is a function of $A[0..i]$, i.e. $B(i) = F.A[0..i]$. The first element of stream $B$, $B(0)$, depends on $A(0)$ only. For $i \geq 0$, we have $B(i+1) = F.A[0..i+1]$. Assume that function $F$ has the property that for some $j$, $0 \leq j \leq i+1$, there exists a function $G$ such that $F.A[0..i+1] = G.(F.A[0..j-1, j+1..i+1]).A(j)$, i.e. $A(j)$ can be split off in a generalized expression. Thus, it may be worthwhile to introduce another process $q$, say, that produces $F.A[0..j-1, j+1..i+1]$ along an output channel $b_q$. If process $p$ produces output stream $B$ along channel $b_p$, we have $b_p(i+1) = G.b_q(i).A(j)$. As will be shown later, specifications of the above structure often give rise to parallel programs that have one process taking care of both (external) input and output. Now, assume that $j \neq i+1$. Process $p$ receives $A(i+1)$ along its input channel and, since $j \neq i+1$, sends this value to process $q$ which in turn produces $b_q(i) = F.A[0..j-1, j+1..i+1]$. Meanwhile process $p$ is idle and waiting for communication $b_q(i)$ needed to compute $b_p(i+1) = G.b_q(i).A(j)$. Clearly, processes $p$ and $q$ do not work in parallel, and, as a consequence, the decision to consider terms depending on $A(j)$, where $j \neq i+1$, in isolation is not a good idea.

The above observation illustrates how one can foresee whether an i/o-relation is an appropriate generalization of the specification or not.

**Heuristic 2.6**

If the problem specification allows a solution with $O(1)$ latency then a good heuristic to obtain i/o-relations is to generalize the specification by splitting off the terms containing the last element of the input stream on which the next output value depends.

For some problems it can be shown that the specification does not admit a solution of $O(1)$ latency. In that case there is no need to split off terms containing the last element of the input stream on which the output depends. Other possibilities for generalizing the specification are shown in the following example.

The OCL problem is a window computation of which the computed result of a window depends on the last element of that window. All window elements are compared with the last value received. Unless the values of the input stream are all within a small range, a solution of latency $O(1)$ is not possible. We can, however, obtain a latency of $O(\log N)$ or $O(N)$. A solution of latency $O(\log N)$ is given in [20] and results in a tree topology. We present a solution of latency $O(N)$.

**Example 2.7** (Occurrence Count Last)

The specification of the OCL problem from Example 2.4 reads: given an input stream $A$ of integers, compute output stream $B$ such that

$$B(i) = (\# j : i - N < j \leq i : A(j) = A(i))$$
for $i \geq 0$. For $N=1$, we have $B(i)=1$. For $N>1$, we derive
\[
B(i) = \begin{cases} \{ \text{specification} \} \\
(\# j : i-N < j \leq i : A(j) = A(i)) \\
N>1 : \text{split off } j=i-N+1 \} \\
A(i-N+1) = A(i) + (\# j : i-N+1 < j \leq i : A(j) = A(i))
\end{cases}
\]
where $[.]$ is defined by $[true] = 1$ and $[false] = 0$.

As in Example 2.5, we observe the similarity between the specification and the quantified expression in the last line of the derivation. We conjecture a linear arrangement of $N$ processes, numbered from 0 onwards. Process $k$ has output channel $b_k$ satisfying
\[
b_k(i) = (\# j : i-N+k < j \leq i : A(j) = A(i))
\]
Note that $B(i) = b_0(i)$ and that $b_0(i) = [A(i-N+1) = A(i)] + b_i(i)$ for $i \geq 0$.

We conclude this section with one more example. In this example, the specification is symmetric in two arguments. We shall, however, not maintain this symmetry when generalizing the specification. It turns out that this decision excludes the construction of an efficient parallel program, as will become clear in the sequel.

**Example 2.8 (Finite Convolution)**
The Finite Convolution problem is a window computation involving two input streams, $A$ and $B$, and one output stream $C$ satisfying
\[
C(i) = (\sum j, h : 0 \leq j \land 0 \leq h \land j+h = N-1 : A(i-j) \ast B(i-h))
\]
where $N$ is the window length and each negatively indexed element of stream A and B has an appropriate value.

The value of $C(i)$ depends on both $A(i-N..i)$ and $B(i-N..i)$. The specification is symmetric in the input streams. Using Heuristic 2.6, the proper thing to do, when generalizing the specification, is to split off those terms in the quantified expression that depend on $A(i)$ and $B(i)$. As an illustration of what happens when symmetry is not maintained we split off only the term that depends on $A(i)$. We derive
\[
C(i+1) = \begin{cases} \{ \text{specification} \} \\
(\sum j, h : 0 \leq j \land 0 \leq h \land j+h = N-1 : A(i+1-j) \ast B(i+1-h)) \\
\text{split off } (j, h) = (0, N-1), \text{ destroying the symmetry} \} \\
A(i+1) \ast B(i+2-N) \\
+ (\sum j, h : 1 \leq j \land 0 \leq h \land j+h = N-1 : A(i+1-j) \ast B(i+1-h)) \\
\text{shift range of quantification} \} \\
A(i+1) \ast B(i+1-(N-1)) \\
+ (\sum j, h : 0 \leq j \land 0 \leq h \land j+h = N-2 : A(i-j) \ast B(i-h+1))
\end{cases}
\]
2.4. Set of Equations

From this derivation we infer that a linear arrangement of processes indexed from 0 up to $N-1$ is appropriate, in which process $k$ has an output channel $c_k$ satisfying i/o-relation

$$c_k(i) = (\Sigma j, h : 0 \leq j \land 0 \leq h \land j + h = N - 1 - k : A(i - j) \ast B(i - h + k))$$

Process 0 establishes $c_0(i) = C(i)$.

The above example shows that it is possible to obtain i/o-relations by destroying symmetry when generalizing a specification. However, having an i/o-relation does by no means indicate that an efficient parallel program can be obtained, as will become apparent later.

Heuristic 2.9

If a problem specification is symmetric in some parameters, then it is attractive to maintain this symmetry when generalizing the specification. The i/o-relations obtained in this way often give rise to efficient parallel programs.

2.4 Set of Equations

In the previous section, i/o-relations were derived by generalizing the specification of a problem. These i/o-relations specify the values that are communicated along internal channels and are considered as (a part of) a specification of the processes that constitute the overall computation. Moreover, the i/o-relations lead to a particular choice of the topology of the process network. In this section, we determine how a process computes the values to be communicated in such a way that the i/o-relations are satisfied.

In Example 2.5, we obtained the following relation between the output channels of process 0 and 1: $b_0(i+2) \equiv (A(0)=A(i+2)) \land b_1(i)$. Apparently, the computation of $b_0(i+2)$ involves the value $b_1(i)$, communicated along an input channel local to process 0, and values $A(0)$ and $A(i+2)$, two elements of the global input stream $A$. This observation leads to the idea of introducing an additional input channel for process 0 establishing the communication of elements from stream $A$ to this process. This yields an additional i/o-relation which specifies the values communicated along that input channel. Since an input channel for one process is an output channel for another process, input and output channels are treated in the same way.

Summarizing the above, starting from an i/o-relation equations are derived that define the values communicated along an output channel in terms of values received along input channels. In the derivation, one often has to introduce additional channels to a process and, hence, additional i/o-relations.

Example 2.10 (Palindrome Recognizer)

For the Palindrome Recognizer we have the following i/o-relation, cf. Example 2.5

$$b_k(i) \equiv (\forall j, h : 0 \leq j \land 0 \leq h \land j + h = i : A(j+k) = A(h+k))$$
Since \( b_k(0) = \text{true} \) and \( b_k(1) = (A(k) = A(k + 1)) \), we derive for \( i \geq 2 \), as in Example 2.5:

\[
\begin{align*}
\quad & b_k(i+2) \quad \\
\equiv & \quad \{ \text{i/o-relation } b_k \} \\
\quad & (\forall j, h : 0 \leq j \land 0 \leq h \land j+h = i+2 : A(j+k) = A(h+k)) \\
\equiv & \quad \{ i \geq 0 : \text{split off } (j,h) = (i+2,0) \text{ and } (j,h) = (0,i+2) \} \\
\quad & (A(i+2+k) = A(k)) \land (A(k) = A(i+2+k)) \\
\quad & \land (\forall j, h : 0 \leq j \land 0 \leq h \land j+h = i+2 : A(j+k+1) = A(h+k)) \\
\equiv & \quad \{ \text{shift range of quantification} \} \\
\quad & (A(k) = A(i+2+k)) \\
\quad & \land (\forall j, h : 0 \leq j \land 0 \leq h \land j+h = i : A(j+(k+1)) = A(h+(k+1))) \\
\equiv & \quad \{ \text{i/o-relation } b_{k+1} \} \\
\quad & (A(k) = A(i+2+k)) \land b_{k+1}(i)
\end{align*}
\]

For the computation of \( b_k(i) \), process \( k \) needs two elements of the global input stream \( A \). One is fixed \( (A(k)) \) and one depends on \( i \) \( (A(i+2+k)) \). For process \( k \), we therefore introduce an input channel \( a_k \) satisfying i/o-relation

\[
a_k(i) = A(i+k)
\]

For this input channel of process \( k \) we have to decide of which process channel \( a_k \) is an output channel. There are two candidates: either process \( k-1 \) or process \( k+1 \). Since \( a_0(i) = A(i) \), the environment can readily provide the input for process \( 0 \). We therefore decide that channel \( a_k \) is an output channel of process \( k-1 \) (see Figure 2.3). For output channel \( a_{k+1} \) of process \( k \) we derive

\[
\begin{align*}
\quad & a_{k+1}(i) \\
\quad & = \quad \{ \text{i/o-relation } a_{k+1} \} \\
\quad & A(i+k+1) \\
\quad & = \quad \{ \text{calculus} \} \\
\quad & A((i+1)+k) \\
\quad & = \quad \{ \text{i/o-relation } a_k \} \\
\quad & a_k(i+1)
\end{align*}
\]

Summarizing, we have derived the following set of equations for the output channels of process \( k \)

\[
b_k(0) \equiv \text{true}
\]
2.4. Set of Equations

\[ b_k(1) \equiv (a_k(0) = a_k(1)) \]
\[ b_k(i+2) \equiv (a_k(0) = a_k(i+2)) \land b_{k+1}(i) \]
\[ a_{k+1}(i) = a_k(i+1) \]

for \( i \geq 0 \).

In the above example, one additional channel has been introduced, viz. an input channel that takes care of distributing the elements of the (external) input stream. Since the i/o-relations obtained from a generalization of the specification are i/o-relations of output channels only, it is obvious that at least one additional input channel for each process had to be introduced.

When introducing an additional channel, one needs to determine which other process is engaged in this communication. In the example of palindrome recognition, we observed that the values communicated along input channel \( a_0 \) could be readily produced by the environment. On account of this observation, we decided that channel \( a_k \) is an output channel of process \( k-1 \).

Heuristic 2.11

When determining which processes are engaged in communications along a channel, it is a good strategy to consider a process at the boundary of the process network and to determine whether this process can readily access or produce the values specified in the i/o-relation.

In the case of Palindrome Recognition, the only process at the boundary of the process network is process 0. As a second example, we consider the OCL problem.

Example 2.12 (Occurrence Count Last)

For the OCL problem we have the following i/o-relation, cf. Example 2.7,

\[ b_k(i) = (\# \, j : i-N+k < j \leq i : A(j) = A(i)) \]

where the processes are indexed from 0 up to \( N-1 \). For \( k=N-1 \) we have \( b_{N-1}(i) = A(i)=A(i) \) and for \( 0 \leq k < N-1 \) we derive

\[ b_k(i) = \{ i/o-relation \, b_k \} \]
\[ \quad = \{ \text{split off } j=i-N+k+1 \} \]
\[ \quad = \{ \text{split off } j=i-N+k+1 \} \]
\[ \quad = \{ i/o-relation \, b_{k+1} \} \]
\[ \quad \quad + \quad A(i) = A(i) \]
\[ \quad \quad + \quad b_{k+1}(i) \]

For the computation of \( b_k(i) \), process \( k \) needs to have the elements \( A(i-N+k+1) \) and \( A(i) \) of the input stream at its disposal. Observe that \( A(i-N+k+1) = A(i) \) for \( k=N-1 \). Therefore, process \( N-1 \) is a good candidate for receiving input stream \( A \).
Chapter 2. Fine-grained Parallelism

Figure 2.4: linear arrangement of processes for the OCI problem

As a consequence, we introduce two additional output channels, $c_k$ and $d_k$, for each process $k$ (see Figure 2.4) satisfying i/o-relations

$$c_k(i) = A(i-N+k)$$
$$d_k(i) = A(i)$$

For output channel $c_k$ we derive

$$c_k(0) = \begin{cases} 
\{ i/o\text{-}relation \ c_k \} \\
A(-N+k) \\
\{ \text{for } k<N: A(-N+k) \text{ is an appropriate value, 0 say } \} 
\end{cases}$$

and for $i \geq 0$

$$c_k(i+1) = \begin{cases} 
\{ i/o\text{-}relation \ c_k \} \\
A(i+1-N+k) \\
\{ \text{calculus} \} \\
A(i-N+(k+1)) \\
\{ i/o\text{-}relation \ c_{k+1} \} \\
c_{k+1}(i)
\end{cases}$$

For output channel $d_k$ we have

$$d_k(i) = \begin{cases} 
\{ i/o\text{-}relation \ d_k \} \\
A(i) \\
\{ i/o\text{-}relation \ d_{k+1} \} \\
d_{k+1}(i)
\end{cases}$$

Summarizing, we have derived the following set of equations for the output channels of process $k$, $0 \leq k < N-1$,

$$b_k(i) = [c_{k+1}(i)=d_{k+1}(i)] + b_{k+1}(i)$$
$$c_k(0) = 0$$
$$c_k(i+1) = c_{k+1}(i)$$
$$d_k(i) = d_{k+1}(i)$$
2.4. Set of Equations

for \( i \geq 0 \).

Notice that in the right-hand side of the expression to calculate \( b_k(i) \) of the example above the indices, viz. \( i \), of the channels are equal. This offers the possibility of treating the channels in a similar way. It is even possible to consider the collection of three output channels as one channel over which triples are communicated.

Some i/o-relations are appropriate for deriving a good set of equations; others are not. As an example of an i/o-relation that is not appropriate we take the i/o-relation of Example 2.8 for Finite Convolution.

**Example 2.13 (Finite Convolution)**

In Example 2.8, we derived the following i/o-relations for the Finite Convolution problem

\[
c_k(i) = (\Sigma j, k : 0 \leq j \land 0 \leq h \land j+h = N-1-k : A(i-j) \ast B(i-h+k))
\]

This problem is a window computation in which the negatively indexed elements of input streams \( A \) and \( B \) have appropriate values. We decide these values to be 0.

Consider the first value communicated along output channel \( c_k \)

\[
c_k(0)
\]

\[
= \{ \text{i/o-relation } c_k \}
\]

\[
(\Sigma j, k : 0 \leq j \land 0 \leq h \land j+h = N-1-k : A(-j) \ast B(-h+k))
\]

\[
= \{ A(-k)=0 \text{ for } k>0 \}
\]

\[
A(0) \ast B(2k-(N-1))
\]

Before producing output \( c_k(0) \), process \( k \) must receive \( A(0) \) and \( B(2k-(N-1)) \). The element of the \( B \)-stream is produced either by process \( k-1 \) or by process \( k+1 \).

First, assume that it is produced by process \( k-1 \). Introduce input channel \( b_k \) that satisfies \( b_k(0) = B(2k-(N-1)) \). Process \( k \) should produce \( b_{k+1}(0) \) but, unfortunately, it cannot do so for all process indices \( k \). Take, for \( k=N-2 \), then \( b_{N-1}(0) = B(N-1) \). However, process \( N-2 \) can have at its disposal \( b_{N-2}(0) = B(N-3) \) only.

Next, assume that \( B(2k-(N-1)) \) is produced by process \( k+1 \). As a consequence, we introduce output channel \( b_k \) that satisfies \( b_k(0) = B(2k-(N+1)) \). Again, take \( k=N-2 \). Process \( N-2 \) should produce \( B(N-5) \), but it can only have at its disposal \( b_{N-1}(0) = B(N-3) \).

We conclude that the i/o-relation does not lead to a good set of equations. As already mentioned before, this i/o-relation was obtained from its specification by violating the heuristic of maintaining symmetry.

\[
\]

**Unspecified values**

When introducing an auxiliary channel \( a_k \) it is not always necessary to derive an i/o-relation in which \( a_k(i) \) is specified for all \( i \). This is the case if for some index \( i \) value \( a_k(i) \) is not needed for any computation. In such a case, we may leave that particular
value \( e_k(i) \) unspecified. The following example shows that it is sometimes useful to have an i/o-relation that is unspecified for certain indices. This example is taken from [27].

**Example 2.14** (Dirichlet Convolution)

In [27] the design of a parallel program for Dirichlet convolution is discussed. The program consists of an unbounded linear arrangement of processes indexed from 0. Each process \( k, \, k \geq 1 \), has, amongst others, input channels \( e_k \) and \( f_k \). Input channel \( f_k \) satisfies \( f_k(i) = F(i+1) \) for all \( i \geq 0 \), where \( F \) is an external input stream. Unlike channel \( f_k \), input channel \( e_k \) is not specified for all indices \( i \). The i/o-relation reads

\[
e_k(i) = F(1 + k \div (i+1))
\]

for \( i \) satisfying \((i+1)^2 > (i+k+1)\).

Process \( k \) provides the values communicated along \( e_{k+1} \). We only need to calculate \( e_{k+1}(i) \) for \( i \) satisfying \((i+1)^2 > (i+k+2)\). Let \((m|n)\) denote 'm is a divisor of n'. We derive

\[
e_{k+1}(i) = \begin{cases} \text{i/o-relation } e_{k+1} \\ F(1 + (k+1) \div (i+1)) \end{cases}
\]

\[
{} = \begin{cases} \neg(i+1 \mid k+1) \implies (k+1) \div (i+1) = k \div (i+1) \\ \text{if } \neg(i+1 \mid k+1) \implies F(1 - k \div (i+1)) \\ \text{if } (i+1 \mid k+1) \implies F(1 + (k+1) \div (i+1)) \end{cases}
\]

\[
{} = \begin{cases} (i+1 \mid k+1) \land (i+1)^2 > (i+k+2) \implies (k+1) \div (i+1) = k \div i, \\ \text{cf. [27]} \end{cases}
\]

\[
{} = \begin{cases} \text{i/o-relation } e_k; (i+1)^2 > i+k+2 > i+k+1 \\ \text{if } \neg(i+1 \mid k+1) \implies e_k(i) \\ \text{if } (i+1 \mid k+1) \land i^2 > i+k \implies e_k(i-1) \\ \text{if } (i+1 \mid k+1) \land i^2 \leq i+k \implies F(1 + (k+1)/(i+1)) \end{cases}
\]

Notice that \( e_k(i-1) \) is only specified for \( i^2 > i+k \). If \((i+1)^2 > (i+k+2) \) and \((i+1 \mid k+1) \), and \( i^2 \leq i+k \) then it turns out that \((k+1)/(i+1) = j-1 \) (cf. [27]). It is allowed to choose a convenient value for each \( e_k(i) \) that is not specified. As a result, we have the following equation for \( e_{k+1} \)

\[
e_{k+1}(i) = \begin{cases} e_k(i) & \text{if } \neg(i+1 \mid k+1) \lor i = 0 \\ e_k(i-1) & \text{if } (i+1 \mid k+1) \land i^2 > i+k \\ f_k(i-1) & \text{if } (i+1 \mid k+1) \land 0 < i^2 \leq i+k \\ f_k(i) & \text{if } (i+1 \mid k+1) \land i^2 > i+k + 1 \end{cases}
\]

for \( i \geq 0 \). **\( \square \)**
Negatively indexed elements

A window computation is also a computation for which output values are not defined for all indices. More specifically, in a computation with window size $N$ the first $N-1$ outputs are not defined. In that case we specify some arbitrary value (we indicated three possibilities, before). We show that it does not matter in what way the first $N-1$ outputs of a window computation are specified.

Consider a window computation with window length $N$. Let $A$ denote the input stream and let output stream $B$ be specified by

$$B(i) = F.A[i-N..i]$$

for $i \geq N-1$. $F$ is some function on a window. As a next step in the design, we generalize this specification in order to obtain i/o-relations. We derive

$$B(i)$$

$$= \{ \text{specification} \}$$

$$F.A[i-N..i]$$

$$= \{ \text{split off } A[i-N+1], \text{ assume some function } g_N \}$$

$$g_N.A[i-N+1].(F.A[i-N+1..i])$$

In the derivation we split off $A[i-N+1]$. We could also have chosen to split off $A(i)$ resulting in $h_N.A(i).(F.A[i-N..i-1])$. This choice is rather arbitrary and would have resulted in a similar program. Notice, furthermore, that the OCL problem does not fit into this scheme (cf. Example 2.7), since in that case function $g_N$ would have an additional argument, viz. $A(i)$. Again, our choice is not an essential simplification of the problem.

We introduce $N$ processes indexed from 0 up to $N-1$. Each process $k$ has an output channel $b_k$ that satisfies i/o-relation

$$b_k(i) = F.A[i-N+k..i]$$

for $i \geq N-1-k$. For process $N-1$ we have $b_{N-1}(i) = F.A(i)$ for all $i \geq 0$. Output value $b_k(i)$ is specified for $i \geq N-1-k$ only. We then have

$$b_k(i)$$

$$= \{ \text{i/o-relation } b_k \}$$

$$F.A[i-N+k..i]$$

$$= \{ \text{split off } A(i-N+k+1) \}$$

$$g_{N-k}.A[i-N+k+1].(F.A[i-N+k+1..i])$$

$$= \{ \text{i/o-relation } b_{k+1}; i \geq N-1-k > N-1-(k+1) \}$$

$$g_{N-k}.A[i-N+k+1].b_{k+1}(i)$$

$$= \{ \text{i/o-relation additional channel } c_{k+1}, \text{ see below } \}$$

$$g_{N-k}.c_{k+1}(i).b_{k+1}(i)$$

In the derivation, output channel $c_k$ has been introduced (cf. Figure 2.5) satisfying
Figure 2.5: linear arrangement of processes for a window computation

\[ c_k(i) = A(i-N+k) \]

for \( i \geq N-k \). Notice that \( c_k(i) = A(i) \) for all \( i \geq 0 \). For \( i \geq N-k \) we have

\[
\begin{align*}
  c_k(i) & = \{ i/o\text{-}relation \ c_k \} \\
  & = \{ \text{calculus} \} \\
  & = A(i-1-N+(k+1)) \\
  & = \{ i/o\text{-}relation \ c_{k+1}; \ i-1 \geq N-(k+1) \}
\end{align*}
\]

\[ c_{k+1}(i-1) \]

Summarizing, for processes \( k \) such that \( 0 \leq k < N-1 \)

\[
\begin{align*}
  b_k(i) & = g_{N-k} \cdot c_{k+1}(i) \cdot b_{k+1}(i) & \text{for } i \geq N-1-k \\
  c_k(i) & = c_{k+1}(i-1) & \text{for } i \geq N-k
\end{align*}
\]

and for process \( N-1 \) we have

\[
\begin{align*}
  b_{N-1}(i) & = F_c c_N(i) & \text{for } i \geq 0 \\
  c_{N-1}(i) & = c_N(i-1) & \text{for } i \geq 1
\end{align*}
\]

For the unspecified indices \( i \) of \( b_k(i) \) and \( c_k(i) \) we are free to choose appropriate values. In this case, the simplest solution is to define \( c_0(0) \) as a constant and to apply the equations for \( b_k(i) \) and \( c_k(i) \) for all other unspecified indices.

In general, occurrences of a value with an unspecified index may even be replaced by different values. Consider, for instance, output channel \( c \) and \( d \) with the following equations for the first value that has to be communicated

\[
\begin{align*}
  c(0) & \equiv (A(-1) \neq C) \\
  d(0) & = A(-1) + 1
\end{align*}
\]

where \( C \) is an (unknown) constant. From the equation for \( c(0) \), we infer that choosing \( A(-1) \neq C \) (or \( A(-1) = C+1 \)) yields \( c(0) \equiv \text{true} \). For \( d(0) \) it is more convenient to choose \( A(-1) = -1 \), resulting in \( d(0) = 0 \). Both replacements of \( A(-1) \) are not consistent with each other, but that is of no importance.

In the section on specifications, three possible solutions for specifying the OCL problem have been presented. We adopted the third possibility (see Example 2.4) in which negatively indexed elements of the input stream are given appropriate values. This solution reflects the solution for the general window problem given above. An
advantage of choosing appropriate values for negatively indexed elements of streams is that specifications can be generalized even if they deal with windows that contain negatively indexed elements. In this way, extensive case analysis can be avoided.

Strategy 2.15
Negatively indexed elements of input streams only affect elements of output streams that are unspecified. In derivations, it is allowed to choose at any time any appropriate value for a negatively indexed element of an input stream. These values do not even have to be chosen consistently.

2.5 Communication Behavior

In the previous section, we discussed how to obtain a set of defining equations for the values communicated along output channels. If an output value is a function of an input value, then that input value should be received before the output value can be produced. As a result, an equation imposes a partial order on the communications along channels. From a collection of partial orders, one for each output channel, the communication behavior of the processes can be determined. Such a communication behavior defines the order in which a process accesses its channels. It is obvious that a communication behavior should be consistent with the partial orders defined by the set of equations.

The communication behavior of the external channels of the program can be a part of the specification. In that case we have as an additional constraint that the overall communication behavior of the program should be consistent with the specified external communication behavior. In the sequel, however, we assume that no external communication behavior has been specified.

Given an appropriate communication behavior of a process, we can compose the program text of that process. Our programs are of a CSP-like style [7] in which communications with other processes are programmed explicitly. We adopt the convention to introduce at least one variable for each input channel. As a rule for constructing a communication behavior, we try to find a communication behavior that introduces minimal buffering.

Example 2.16 (Palindrome Recognizer)
The set of equations for output channels $b_k$ and $a_{k+1}$ of process $k$ of the Palindrome Recognizer problem is (cf. Example 2.10)

\[
\begin{align*}
  b_k(0) &\equiv \text{true} \\
  b_k(1) &\equiv (a_k(0) = a_k(1)) \\
  b_k(i+2) &\equiv (a_k(0) = a_k(i+2)) \land b_{k+1}(i) \\
  a_{k+1}(i) &\equiv a_k(i+1)
\end{align*}
\]

for $i \geq 0$. For channel $b_k$ we have $a_k(0) < b_k(1) < a_k(1) < b_k(i+1) < b_k(i+1)$ for $i \geq 0$. This yields the following communication behavior (with minimal buffering).

\[
\text{CB}(b_k) : a_{k}; b_k; a_{k+1}(b_k; a_k, b_{k+1})^*
\]
For channel \( a_{k+1} \) we obtain
\[
\text{CB}(a_{k+1}) : a_k; (a_k; a_{k+1})^*
\]
Combining \( \text{CB}(b_k) \) and \( \text{CB}(a_{k+1}) \) into an overall communication behavior yields
\[
\text{CB}_k : a_k; b_k; a_k; (b_k, a_{k+1}; a_k, b_{k+1})^*
\]
Since \( \text{CB}_k \{\{a_k, b_k, b_{k+1}\} = \text{CB}(b_k) \) and \( \text{CB}_k \{\{a_k, a_{k+1}\} = \text{CB}(a_{k+1}) \), this communication behavior is consistent with \( \text{CB}(b_k) \) and \( \text{CB}(a_{k+1}) \). The overall communication behavior has minimal buffering.

Remark: in communication behavior \( \text{CB}_k \) there is an ordering of communications along channels \( a_{k+1} \) and \( b_{k+1} \): \( a_{k+1} b_k < b_{k+1} a_k \). A communication behavior that is consistent with \( \text{CB}(b_k) \) and \( \text{CB}(a_{k+1}) \) and in which \( a_{k+1} \) and \( b_{k+1} \) are not ordered is
\[
\text{CB}'_k : a_k; b_k; a_k; (a_k; a_{k+1}; b_{k+1}; b_k)^*
\]
However, in \( \text{CB}'_k \) communications along \( a_{k+1} \) and \( b_k \) are ordered.

Communication behavior \( \text{CB}_k \) of the previous example is consistent with the set of equations derived in Example 2.10. There is, however, another correctness concern, viz. the absence of deadlock. As already mentioned in Section 2.1, we require that for all processes \( s \) and \( t \)
\[
\text{CB}_p \{S_{st} = \text{CB}_p \{S_{st}
\]
where \( \text{CB}_p \) denotes the communication behavior of process \( p \) and \( S_{p+} \) denotes the set of channels that process \( p \) and \( q \) share.

Example 2.17 (PalindromeRecognizer)
For process \( k \) we have (cf. Example 2.16)
\[
\text{CB}_k : a_k; b_k; a_k; (b_k, a_{k+1}; a_k, b_{k+1})^*
\]
Process \( k \) and process \( k+1 \) share channels \( a_{k+1} \) and \( b_{k+1} \). Projection on shared channels yields
\[
\text{CB}_k \{a_{k+1}, b_{k+1}\} = (a_{k+1}; b_{k+1})^*
\]
and
\[
\text{CB}_{k+1} \{a_{k+1}, b_{k+1}\} = a_{k+1}, b_{k+1}; (a_{k+1}; b_{k+1})^*
\]
We cannot conclude deadlock-freedom. The problem in \( \text{CB}_k \) is that \( a_k(0) \) and \( b_k(0) \) are not ordered whereas \( a_{k+1}(0) \) and \( b_{k+1}(0) \) are. We solve this problem by imposing an ordering on \( a_k(0) \) and \( b_k(0) \) and obtain communication behavior
\[
\text{CB}'_k : a_k; b_k; a_k; (b_k, a_{k+1}; a_k, b_{k+1})^*
\]
Now, we have: \( \text{CB}_k \{a_{k+1}, b_{k+1}\} = \text{CB}'_{k+1} \{a_{k+1}, b_{k+1}\}. \)
2.5. Communication Behavior

In this example, communication behavior CB_k has been transformed into CB'_k by adding an additional ordering on communications. Notice that this transformation is such that communication along an input channel has been advanced and as a result communication along an output channel has been delayed.

In general, by advancing input and delaying output the communication behavior of a process remains consistent with the partial order on communications given by the set of defining equations. In the construction of a communication behavior we aim at minimal buffering. If, however, this behavior has to be transformed by delaying output or advancing input, buffering of input values is often introduced.

Example 2.18 (Occurrence Count Last)

For the output channels of process k (k≠N−1) of the OCL problem we have the following set of equations (see Example 2.12)

\[
\begin{align*}
 b_k(i) &= [c_{k+1}(i)=d_{k+1}(i)] + b_{k+1}(i) \\
 c_k(0) &= 0 \\
 c_k(i+1) &= c_{k+1}(i) \\
 d_k(i) &= d_{k+1}(i)
\end{align*}
\]

for \(i \geq 0\). (For process N−1, we have \(c_{N−1}(i+1)=d_N(i)\) and \(b_{N−1}(i)=1\).) From these equations, we obtain the following communication behaviors for channels \(b_k\), \(c_k\), and \(d_k\):

\[
\begin{align*}
 CB(b_k) & : (b_{k+1}; c_{k+1}; d_{k+1}; b_k)^* \\
 CB(c_k) & : (c_{k+1})^* \\
 CB(d_k) & : (d_{k+1}; d_k)^*
\end{align*}
\]

These communication behaviors result in an overall communication behavior CB_k of process k that has minimal buffering:

\[
CB_k : c_k; (b_{k+1}; c_{k+1}; d_{k+1}; b_k) (c_k; d_k)^*
\]

This communication behavior is not satisfactory since

\[
CB_k \{b_{k+1}, c_{k+1}, d_{k+1}\} = (b_{k+1}, c_{k+1}, d_{k+1})^*
\]

and

\[
CB_{k+1} \{b_{k+1}, c_{k+1}, d_{k+1}\} = c_{k+1} : (b_{k+1}, c_{k+1}, d_{k+1})^*
\]

This problem can be solved by delaying output \(c_k\), resulting in

\[
CB'_k : (b_{k+1}, c_{k+1}, d_{k+1}; b_k, c_k, d_k)^*
\]

From \(CB'_k \{c_k, c_{k+1}\} = (c_{k+1}; c_k)^*\) we infer that channel \(c_{k+1}\) is buffered. (For the sake of completeness, we mention that the communication behavior of process N−1 is \((d_N; b_{N−1}, c_{N−1}, d_{N−1})^*\).)
In Example 2.17 and 2.18 communication behaviors were constructed in which there is an alternation between input and output actions. Since channels are input channels to one process and output channels to another process, alternation of input and output by a process causes neighboring processes to execute "out of phase". If a process performs an input action, its neighboring processes performs output actions, and vice versa. This alternation between input actions and output actions often results in programs where neighboring processes access shared channels in the same way, thus avoiding deadlock.

**Heuristic 2.19**
When constructing an overall communication behavior for a process, it is a good strategy to search for a communication behavior that is an alternation of input actions and output actions.

From the set of defining equations together with a consistent communication behavior it is easy to code the program text of each process. Apart from the declaration of variables, the program text consists of the communication behavior to which computations are added.

**Example 2.20 (Palindrome Recognizer)**
For Example 2.10 we derived the following set of equations for output channels $b_k$ and $a_{k+1}$

$$
\begin{align*}
  b_k(0) & \equiv \text{true} \\
  b_k(1) & \equiv (a_k(0) = a_k(1)) \\
  b_k(i+2) & \equiv (a_k(0) = a_k(i+2)) \land b_{k+1}(i) \\
  a_{k+1}(i) & = a_k(i+1)
\end{align*}
$$

resulting in communication behavior (cf. Example 2.17)

$$
\text{CB}_k : a_k; b_k; a_k; (b_k, a_{k+1}; a_k, b_{k+1})^*
$$

For each input channel a variable is introduced. Furthermore, one variable is needed for $a_k(0)$, since this value is used for each output value $b_k(i)$. As a result, three variables are introduced for process $k$, giving the following program text.

```plaintext
\{ va0, va : symbol; vb : bool;
  a_k?va;
  va0, vb := va, true
  b_k!vb
  a_k?va;
  vb := (va0==va)
  \{ b_k!vb, a_{k+1}?va
    ; b_{k+1}?vb, a_k?va
    ; vb := vb \land (va0==va)
  \}^*
\}
```

\[ \square \]
2.6. Performance

In this example, the program requires one 'local' variable, viz. \( \nu \). The other variables, \( \nu_0 \) and \( \nu_b \), are needed for receiving values communicated along input channels. If a communication behavior requires buffering of a channel additional variables have to be introduced.

**Example 2.21** (Occurrence Count Last)

For process \( k \) \((k \neq N-1)\) of the OCL problem we have the following set of equations (cf. Example 2.12)

\[
\begin{align*}
    b_k(i) &= [c_{k+1}(i) = d_{k+1}(i)] + b_{k+1}(i) \\
    c_k(0) &= 0 \\
    c_k(i+1) &= c_{k+1}(i) \\
    d_k(i) &= d_{k+1}(i)
\end{align*}
\]

and the following communication behavior is which channel \( c_{k+1} \) in buffered (see Example 2.18)

\[
CB_k : (b_{k+1}, c_{k+1}, d_{k+1}; b_k, c_k, d_k)^*
\]

There are three input channels one of which is buffered. As a result, the program for process \( k \) requires 4 variables, resulting in

\[
\begin{array}{l}
    \nu_0, \nu_1, \nu_2, \nu_3 : \text{int} \\
    \nu_0 := 0 \\
    \nu_1 := \nu_0 \\
    \nu_2 := \nu_0 \\
    \nu_3 := \nu_0 \\
\end{array}
\]

If it is allowed to communicate unspecified values then the initialization of variable \( \nu_0 \) may be omitted on account of Strategy 2.15 and Example 2.12.

\[ \square \]

2.6 Performance

In the previous sections, we discussed a number of heuristics for designing parallel programs satisfying a specification. The specification describes the relation between input and output of a program. Sometimes the specification of the problem also contains performance requirements that have to be met. We restrict ourselves to two aspects of performance, viz. speed requirements and memory requirements.

A number of aspects of a (parallel) program have to be considered when analyzing speed, i.e. how fast can a program compute its output. First of all, we need to know the complexity of the computations carried out by the individual processes. Secondly, we need to know the interaction between the processes. Interaction between processes can cause delays, since a process may be ready to receive an input value along a channel while the sending process is not ready to send that value yet, or vice versa.
This aspect is captured by the communication behavior of the processes. Finally, it is important to know how the program is implemented. Consider, for example, an implementation on a processor network. It makes a difference whether a process has to share a processor with other processes or if a process is the only process allocated to its processor. In the latter case, which we assume for the remainder of this chapter, each process can utilize the full capacity of its processor.

With respect to memory requirements we are primarily interested in the data size of a program (how many variables of a certain type the program uses). The code size of a program will not be considered.

We consider two possibilities for implementing parallel programs. As already mentioned, a program can be implemented on a processor network. The other possibility is the implementation as a VLSI circuit.

When implementing a program on a processor network, we are mainly interested in sheer speed. Of course, the data size of a program must not exceed the available memory, but the memory requirement can be considered as a boundary condition under which speed is optimized.

In an implementation as a VLSI circuit, chip area is an important design restriction. Chip area heavily depends on the data size of the program [29], which is a justification for not considering the code size of a program. When implementing a program as a VLSI circuit, speed requirements can be viewed as a boundary condition under which data size has to be minimized.

**Example 2.22 (Occurrence Count Last)**
Consider the program for the OCL problem given in Example 2.21. We neglect the initialization (which is simply \( v_{ec} := 0 \)). Let \( \tau_k \) denote the only type of computation that is performed by process \( k \), viz. \( 'vb, v_{ec} := v_{b} + [v_{ec} = v_{b}], v_{ec}, v_{c}' \). On account of the symmetry between channels \( b_k, c_k, \) and \( d_k \), we only consider channel \( d_k \). Now, process \( k \) of the OCL problem has the following structure

\[
(d_{k+1}; \tau_k; d_k)^k
\]

for \( 0 \leq k < N \), where \( \tau_{N-1} \) denotes \( 'vb, v_{ec} := 1, v_{ec}, v_{c}' \).

We construct a sequence function \( \sigma \) to analyze the computational complexity of the designed program for the OCL problem. Assume that a (concurrent) communication action takes time \( \alpha \) and that a computation \( \tau_k \) takes time \( \beta \). (Although \( \tau_{N-1} \) is slightly simpler than \( \tau_k \) (\( k \neq N-1 \)), we take the same time complexity for each \( \tau_k \).) From the structure of the processes, sequence function \( \sigma \) should satisfy

\[
\sigma(d_{k+1}, 0) \geq \alpha \\
\sigma(d_{k+1, i+1}) - \sigma(d_{k, i}) \geq \alpha \\
\sigma(d_{k, i}) - \sigma(d_{k+1, i}) \geq \alpha + \beta
\]

for \( i \geq 0 \) and \( 0 \leq k < N \). On account of the third inequality, the first inequality can be simplified to \( \sigma(d_N, 0) \geq \alpha \). The 'strongest' solution for \( \sigma \) is

\[
\sigma(d_k, i) = \alpha(1 + N-k + 2i) + \beta(N-k+i)
\]

which is indeed the best solution, since \( \sigma(d_{k+1}, i+1) - \sigma(d_k, i) = \alpha \) and \( \sigma(d_k, i) - \sigma(d_{k+1}, i) = \alpha + \beta \).

\[\square\]
2.6. Performance

From the sequence function of a program, we can infer performance related issues, since it is known at what time external input and output takes place. Concepts that are relevant are, amongst others, response time, latency, speedup, and efficiency.

In the OCL problem, we have a constant response time of $2\alpha + \beta$. Furthermore, output value $d_{0}(i)$ depends on input values $d_{N}(i-N+1)$ up to $d_{N}(i)$, of which $d_{N}(i)$ is the last one. Hence, we have a latency of $\sigma(d_{0}, i) - \sigma(d_{N}, i) = N(\alpha + \beta)$. As expected, this is a latency of $O(N)$, independent of $i$.

The speedup of a program is a measure for the gain obtained using a parallel program to solve a problem instead of a sequential program. For a given problem instance, speedup expresses how much faster the parallel program is compared to a sequential program that solves the same problem. For a fair comparison between parallel and sequential programs it is important that all programs are executed on the same type of processor. Now, consider the OCL problem in which $L$ output values have to be produced. The production of the $L$-th output takes place at time $\sigma(d_{0}, L-1) = \alpha + N(\alpha + \beta) + (L-1)(2\alpha + \beta)$. A sequential program that is based on the same observations as the parallel program takes time $N\beta$ for each output that has to be produced, resulting in an overall time of $LN\beta$. For large $L$ ($L \gg N$), the speedup — ratio between sequential and parallel execution time — of the parallel program is

$$\frac{LN\beta}{\alpha + N(\alpha + \beta) + (L-1)(2\alpha + \beta)} \rightarrow \frac{LN\beta}{L(2\alpha + \beta)} = \frac{N}{(2\alpha/\beta) + 1}$$

As opposed to the above mentioned sequential program which is based on the same observations and operations as the parallel program, it is more realistic to compare the parallel program to the best sequential program that exists. For the OCL problem, the best sequential program typically maintains a window of input stream $A$ in a sorted way. This opens the possibility for a computation that takes time $2\beta \log N$ for each output value, resulting in an overall time of $2L\beta \log N$. In this approach the speedup of the parallel program is

$$\frac{2L\beta \log N}{\alpha + N(\alpha + \beta) + (L-1)(2\alpha + \beta)} \rightarrow \frac{2L\beta \log N}{L(2\alpha + \beta)} = \frac{2\log N}{(2\alpha/\beta) + 1}$$

The efficiency of a parallel program relates speedup to the number of processors needed by the program. If a program that is executed on $P$ processors yields speedup $s$, then the efficiency of the program is $s/P$. An efficiency of 1 indicates that all processors are maximally utilized and that no overhead (w.r.t. the sequential program) is introduced by the parallel program. The parallel program for the OCL problem consists of $N$ processes and, as a consequence, is executed on $N$ processors. This yields an efficiency of (taking the first expression for speedup)

$$\left(\frac{N}{(2\alpha/\beta) + 1}\right)/N = \frac{1}{(2\alpha/\beta) + 1}$$

Note that an efficiency of 1 is obtained for $\alpha = 0$.

The program for the OCL problem needs $N$ processors, independent of the number of output values that have to be produced. For the Palindrome Recognizer, the number of processors needed depends on the number of output values that have to be produced.
Example 2.23 (Palindrome Recognizer)
Consider the program for the palindrome recognition problem given in Example 2.20. Let \( \tau_k \) denote the only type of computation that process \( k \) performs. Apart from the first computation, \( \tau_k \) denotes \( "vb := vb \land (va = va)" \). Process \( k \) has the following structure

\[
\sigma(a_k; b_k; a_k; \tau_k; (b_k, a_{k+1}; b_{k+1}, a_k; \tau_k))
\]

As in Example 2.22, we take time \( \alpha \) for a (concurrent) communication action and time \( \beta \) for a computation \( \tau_k \). Sequence function \( \sigma \) for this program should satisfy

\[
\begin{align*}
\sigma(a_k, 0) & \geq \alpha \\
\sigma(a_k, i+1) - \sigma(a_k, i) & \geq 2\alpha + \beta \\
\sigma(b_k, i) - \sigma(a_k, i) & \geq \alpha + \beta \\
\sigma(a_{k+1}, i) & = \sigma(b_k, i+1) \\
\sigma(b_{k+1}, i) & = \sigma(a_k, i+2)
\end{align*}
\]

for \( i \geq 0 \) and \( k \geq 0 \). The first inequality may be replaced by \( \sigma(a_0, 0) \geq \alpha \). From these requirements for \( \sigma \) we obtain

\[
\begin{align*}
\sigma(a_k, i) & = \alpha(1+2i+3k) + \beta(2i+3k) \\
\sigma(b_k, i) & = \alpha(2+2i+3k) + \beta(1+2i+3k)
\end{align*}
\]

We then have \( \sigma(a_k, i+1) - \sigma(a_k, i) = 2\alpha + 2\beta \) and \( \sigma(b_k, i) - \sigma(a_k, i) = \alpha + \beta \). Thus, not all inequalities are replaced by equalities, which is, in fact, impossible for this problem.

Between communications \( a_k(i) \) and \( a_k(i+1) \) \( (2\alpha + 2\beta) \) time elapses, whereas process \( k \) itself needs only \( (2\alpha + \beta) \) time to perform the necessary communications and computations. During the remaining time, \( \beta \), process \( k \) is idle and waiting for a communication with a neighbor process which is computing the values to be communicated.

Given sequence function \( \sigma \), we can compute response time, latency, speedup, and efficiency of the parallel program for palindrome recognition.

With respect to the response time we consider external channels \( a_0 \) and \( b_0 \). Since \( \sigma(a_0, i+1) - \sigma(a_0, i) = \sigma(b_0, i+1) - \sigma(b_0, i) = 2\alpha + 2\beta \), we have a constant response time of \( 2\alpha + 2\beta \).

In this program for palindrome recognition, value \( a_0(i) \) is the last input value on which output value \( b_0(i) \) depends. Therefore, the latency of the program equals \( \sigma(b_0, i) - \sigma(a_0, i) = \alpha + \beta \).

In order to determine speedup and efficiency, one needs to know the execution time of a sequential program for palindrome recognition. For now, we do not take the best sequential program that exists, but we consider a sequential program based on the parallel program. For output \( B(i) \), i.e., the recognition of a palindrome of length \( i+1 \), \( (i+1) \text{ div } 2 \) comparisons have to be performed, which takes time \( \beta(i+1) \text{ div } 2 \). Thus, the production of \( L \) output values takes time \( \sum_{i=0}^{L} (i+1) \text{ div } 2 \beta \approx \frac{1}{2}L^2 \beta \). The parallel execution time equals \( \sigma(b_0, L-1) = L(2\alpha + 2\beta) - \beta \). Hence, we obtain speedup (for large \( L \))

\[
\frac{\frac{1}{2}L^2 \beta}{L(2\alpha + 2\beta) - \beta} \approx \frac{L}{8} \frac{\beta}{\alpha + \beta} - \frac{L}{8} \frac{1}{\alpha/\beta + 1}
\]
2.6. Performance

For determining the efficiency of the program, we need to know how many processors participate in the production of $L$ output values. For recognition of the last palindrome $L$ mod 2 comparisons have to be computed. Since each comparison takes place in a different processor, $\frac{1}{2}L$ processors are involved in the computation of the final output. With this result, the efficiency of the program is

$$\left(\frac{L}{8 \cdot (\alpha/\beta) + 1}\right)^{1/2} L = \frac{1}{4 \cdot (\alpha/\beta) + 1}$$

An optimal efficiency of $\frac{1}{4}$ is achieved when the communication time may be neglected with respect to computation time (i.e. when $\alpha/\beta \approx 0$). The fact that an efficiency exceeding $\frac{1}{4}$ cannot be achieved is not surprising: firstly, since, on the average each processor is only active half of the time, viz. process 0 is active during the entire computation, whereas process numbered $\frac{1}{2}L$ performs one or two outputs only, and secondly, since, as already mentioned, each process is idle when a neighbor process performs a computation.

\[\square\]

Some Optimizations

From Example 2.23 above, we infer that idle time caused by the fact that neighboring processes perform computations in an alternating way reduces efficiency by a factor of 2. Reducing idle time by adjusting the programs of the processes is sometimes possible. In this section, we present two possible solutions for reducing idle time in programs of a similar structure as the Palindrome Recognizer. More specifically, we consider processes with a program text for process $k$ of the form

$$a_k?; (\tau_k; b_k!, a_{k+1}!; a_k?, b_{k+1}?)^*$$

where $\tau_k$ denotes the computation of output values.

Optimization 1:

Assume that $\tau_k$ consists of a computation for both the output value of channel $b_k$ and the output value of channel $a_{k+1}$. These computations are denoted by $\tau_k^b$ and $\tau_k^a$, respectively. If shared variables of both computations are only read and computation $\tau_k$ can be replaced by computations $\tau_k^b$ and $\tau_k^a$ in parallel, then the program text of process $k$ may be rewritten as

$$a_k?; (\tau_k^b; b_k!), (\tau_k^a; a_{k+1}!; a_k?, b_{k+1}?)^*$$

The difference with the original program is that the original computation is split into two computations that may be executed in parallel. In a VLSI implementation, both computations can actually be executed in parallel.

Let $\beta_a$ and $\beta_b$ denote the execution time of $\tau_k^a$ and $\tau_k^b$, respectively, such that $\beta = \beta_a + \beta_b$. A sequence function $\sigma$ for the program should satisfy:

$$\sigma(a_k, 0) \geq 0$$
\[ \sigma(a_k, i+1) - \sigma(a_{k+1}, i) \geq \alpha \]
\[ \sigma(a_k, i+1) - \sigma(b_k, i) \geq \alpha \]
\[ \sigma(b_{k+1}, i) - \sigma(a_{k+1}, i) \geq \alpha \]
\[ \sigma(b_{k+1}, i) - \sigma(b_k, i) \geq \alpha \]
\[ \sigma(a_{k+1}, i+1) - \sigma(b_{k+1}, i) \geq \alpha + \beta_k \]
\[ \sigma(b_k, i) - \sigma(a_k, i) \geq \alpha + \beta_k \]
\[ \sigma(b_k, i+1) - \sigma(b_{k+1}, i) \geq \alpha + \beta_k \]

For \( \sigma \), we obtain

\[ \sigma(a_k, i) = \alpha(2i+k+1) + \beta_k(i+k) + \beta_k i \]
\[ \sigma(b_k, i) = \alpha(2i+k+2) + \beta_k(i+k) + \beta_k(i+1) \]

for \( i \geq 0 \) and \( k \geq 0 \). From these equations, we infer that an output is produced every \( 2\alpha+\beta \) time units. The original program produces one output every \( 2\alpha+2\beta \) time units. Hence, idle time is reduced and, thereby, efficiency is increased.

If \( \tau_k \) consists only of a computation for output channel \( b_k \), i.e. \( \tau_k^o = \tau_k \), the above observation is also applicable to processor network implementations. The program for process \( k \) reads

\[ a_k^2, ; ((a_k, b_k)! ; a_{k+1}^2, b_{k+1}^! ?)^* \]

For this program an optimal efficiency of \( \frac{1}{2} \) can be obtained.

**Optimization 2:**

In the original program text, computation \( \tau_k \) is preceded by input statements and it performs a computation on the input values received. Sometimes, however, \( \tau_k \) can be split into two computations \( \tau_k^o \) and \( \tau_k^p \) such that \( \tau_k = \tau_k^o ; \tau_k^p \) and only \( \tau_k^o \) performs computations on the input values received. This assumption makes the following alternative possible

\[ \tau_k^o ; a_k^2, ; (\tau_k^p ; b_k^!, a_{k+1}^2, b_{k+1}^! ? ; \tau_k^o ; a_k^2, b_{k+1}^?)^* \]

When neighbor processes of process \( k \) are executing \( \tau_{k-1}^o \) and \( \tau_{k+1}^o \), process \( k \) executes \( \tau_k^o \) and is not idle. Let \( \beta^o \) and \( \beta^p \) denote the execution times of \( \tau_k^o \) and \( \tau_k^p \), respectively. As a result, \( \beta^o + \beta^p \). If \( \beta^o + \beta^p \) then efficiency is increased by a factor of 2.

Although these optimizations reduce idle time, efficiency is only increased by at most a factor of 2.
Chapter 3

Granularity

In the previous chapter, heuristics for designing fine-grained parallel programs have been discussed. A fine-grained parallel program typically consists of a large number of processes that each perform simple computations. Such a computation consists of a repetition in which communications and computations alternate. From the performance analysis of Section 2.5 it follows that speedup and efficiency of a program depend on the ratio between the time spent on communicating and the time spent on computing. We refer to this ratio as the communication overhead of a program. The larger the communication overhead, the less efficient a program is.

In this chapter, we consider the implementation of parallel programs on a processor network. In processor network implementations it is important to obtain a large speedup and a high efficiency, since this is the main reason for introducing parallelism. It is, therefore, the objective to achieve an efficiency close to 1. The efficiency of a program not only depends on the program itself, i.e. on how much overhead is introduced by the parallel program with respect to its sequential counterpart, but it also depends on characteristics of the processor network on which the program is executed. Take, for instance, a pipe-line program like the program for the OCL problem. If each process is allocated to its own processor, then the efficiency of the program is (cf. Section 2.6)

\[
\frac{1}{(2\alpha/\beta) + 1}
\]

The term \(2\alpha/\beta\) in the denominator of this expression is the communication overhead. Both parameters \(\alpha\) and \(\beta\) depend on characteristics of the processor network on which the program is executed. If communication takes much time in comparison with computation time, i.e. if the communication overhead is large, then the efficiency is small.

When analysing the efficiency of a program one should know what makes the execution of a parallel program different from the execution of a sequential program [5, 31]. First of all, a process of a parallel program not only performs computations but it also performs communications with neighbor processes. Each communication executed is overhead with respect to a sequential implementation of the same program. A second reason why an efficiency close to 1 is not obtained, is that processes in a parallel program become idle when neighbor processes are not ready to engage in a
communication. Finally, a third reason is that, if each process is allocated to its own processor, a processor does not perform any computation when communication of data takes place and, as a consequence, is idle.

Idle time of a processor can be reduced by allocating more than one process to each processor. In that case, a processor schedules another process when the current process is idle (due to either communicating or waiting for communication). In Section 3.1, a global complexity analysis of this solution is presented.

In Section 3.2, techniques for reducing the time spent on communication are discussed. Basically, two methods can be applied. Firstly, the number of communication statements in a program can be reduced by composing larger processes out of a number of (fine-grained) processes. Secondly, considering the fact that a communication between processes requires an initial setup time and a time for the actual data transfer, it is advantageous to compose large messages out of a number of small messages, since in that case a number of communication setups can be omitted.

### 3.1 Multiple Processes per Processor

In this section, processor network implementations are discussed in which more than one process is allocated to each processor. Thereby, idle time of a process does in general not result in idle time of the processor, since another process can take control. For instance, a process of the program for Palindrome Recognition (see Example 2.23) is idle when its neighbor processes are performing a computation. Mapping two neighbor processes on a single processor will increase the efficiency by a factor of 2 (this technique is known as 'passive clustering' [1]).

From a practical point of view it is clear that, in general, multiple processes have to be allocated at a single processor, since the number of processors in a processor network is fixed whereas the number of processes of a program depends on the problem that is solved.

To be able to analyze the speedup and efficiency of a parallel program of which multiple processes are allocated at a single processor, we assume the following model for processor networks. A processor network consists of a number, \( P \) say, of nodes (processors) containing a data processor DP and a communication processor CP. Computations of processes are executed by the DP. Communication statements of processes are executed by both the DP, for setting up the communication, and the CP, for the actual data transfer between nodes. When the CP performs transfer of data, the DP can, independently, execute computations of other processes.

A few assumptions are made to simplify the complexity analysis below. A process switch is considered to be a fast operation that can be neglected in the analysis. This assumption is realistic, since the processors are especially designed for operating in a processor network. Links between CPUs are usually shared by a number of channels between processes. We assume that the overhead introduced by this can be neglected. Furthermore, it is assumed that processes allocated at the same processor operate independently of each other. This means that, for instance, neighbor processes are not allocated at the same processor.
3.1. Multiple Processes per Processor

As an example of a derivation of the efficiency of a network we consider pipe-lined programs, which have a structure of, for instance, the OCL problem (cf. Example 2.22), viz.

\[(d_{k+1}; \tau_k; d_k)^*\]

for \(0 \leq k < N\), where \(\tau_k\) denotes a computation and \(d_k\) and \(d_{k+1}\) denote communication actions. Let the problem size \(N\) be a multiple of \(P\). Assume that processes are evenly distributed over the processor network such that each processor executes \(q=N/P\) processes. For a single computation the DP needs time \(\beta\) and for a single computation the DP and CP need time \(\alpha_0\) (setup time) and \(\alpha_1\), respectively. Let \(\alpha\) denote \(\alpha_0 + \alpha_1\).

The sequential execution time for computing \(L\) output values equals \(t_s = N L \beta\). From Section 2.8, we infer that executing one cycle of the program takes time \(2\alpha + \beta\). As a result, the fraction of the cycle time that the DP is occupied is

\[f_d = \frac{2\alpha_0 + \beta}{2\alpha + \beta}\]

and the fraction of the cycle time the CP is occupied is

\[f_c = \frac{2\alpha_1}{2\alpha + \beta}\]

Note that, in this case, \(f_c + f_d = 1\). In the analysis, we identify three cases: \(q*f_d \geq 1\), \(q*f_c \geq 1\), and \(q*f_c < 1 \land q*f_d < 1\).

\(q*f_d \geq q*f_c \geq 1:\)

Since \(f_c \leq f_d\), the DP is fully occupied and communication is not a bottleneck. The CP is able to keep up its processing with the DP. Each process executes \(L\) cycles that occupy the DP for time \(2\alpha_0 + \beta\) per cycle. Since \(q\) processes share a single processor the parallel execution time of the program equals

\[t_p = qL(2\alpha_0 + \beta)\]

resulting in a speedup of

\[
\frac{t_s}{t_p} = \frac{N L \beta}{q L (2\alpha_0 + \beta)} = \frac{N \beta}{q (2\alpha_0 + \beta)} = P \frac{\beta}{2\alpha_0 + \beta}
\]

which yields communication overhead \(2q\alpha_0\).

\(q*f_c \geq q*f_d \geq 1:\)

Since \(f_d \leq f_c\), the CP is the bottleneck. In each cycle a process executes the CP is occupied for time \(2\alpha_1\); resulting in a parallel execution time of

\[t_p = qL(2\alpha_1)\]

and a speedup of

\[
\frac{t_s}{t_p} = \frac{N L \beta}{q L (2\alpha_1)} = \frac{N \beta}{q 2\alpha_1} = P \frac{\beta}{2\alpha_1}
\]
\[ q \cdot f_c < 1 \land q \cdot f_d < 1: \]

Both the DP and the CP are not fully occupied. Since \( q = q \cdot (f_c + f_d) < 2 \), only one process per processor is allocated. This case has been examined in Section 2.6 where we derived a speedup of

\[ P \cdot \frac{\beta}{2\alpha + \beta} \]

and, hence, communication overhead \( \frac{\beta}{2\alpha + \beta} \).

From the fact that \( f_c \leq f_d \) (≥) implies \( \frac{\beta}{2\alpha + \beta} \geq \frac{\beta}{2\alpha_1} \) (≤), we conclude that, if more than one process is allocated per processor (the first and second case above), an efficiency of

\[ \min\left( \frac{\beta}{2\alpha_0 + \beta} , \frac{\beta}{2\alpha_1} \right) \]

is achieved. This efficiency is better than in the single process per processor case, but it is still close to zero if the communication overhead is large.

### 3.2 Communication Overhead

In this section, techniques for reducing the communication overhead of programs are discussed. Communication overhead expresses the ratio between communication time and computation time of a program and, to a large extent, it determines the efficiency of a program. Communication overhead can be reduced by reducing communication time and by increasing the number of computations executed between successive communications, or by both. Reducing the communication overhead yields programs of a coarser grain.

We discuss three techniques of which the last one is a generalization of the first and the second technique. For this general technique a complexity analysis is presented both for the single process per processor case and for the multiple processes per processor case.

#### 3.2.1 A Technique for Enlarging the Grain Size of Programs

A first technique for enlarging the grain size of a program is to compose larger processes from a number, \( M \) say, of fine-grained processes. Then some of the communications between the original fine-grained processes become internal: these communication actions are replaced by assignment statements. As a consequence, the number of computations between successive communications increases, typically by a factor of \( M \). Since the communication overhead is inversely proportional to this number, the overhead is reduced by a factor of \( M \).

As a second technique for enlarging the grain size of a program, a number \( K \), say, of short messages are combined into larger messages. Then \((K-1)\) communication setup times are saved when communicating a single message of \( K \) values. Applying this
3.2. Communication Overhead

Both techniques are, in fact, special cases of a general technique in which \( M \) fine-grained processes are composed into one large process and \( K \) messages of the fine-grained program are composed into one single message. We refer to this technique as an \([M, K]\)-transformation. When composing larger messages it is often required to introduce larger processes as well. In that case parameter \( K \) should be a divisor of parameter \( M \).

In terms of \([M, K]\)-transformations, the first technique mentioned above is an \([M, 1]\)-transformation and the second technique is a \([K, K]\)-transformation.

3.2.2 Complexity Analysis for the \([M, K]\)-transformation

We consider an \([M, K]\)-transformation of a fine-grained program in which each computation takes time \( \beta \) and each communication takes time \( \alpha = \alpha_0 + \alpha_1 \), where \( \alpha_0 \) is the communication setup time and \( \alpha_1 \) is the complexity of the actual data transfer. Then each computation of the coarse-grained program takes time \( MK\beta \), since for each of the \( K \) output values \( M \) computations are performed. Furthermore, transfer of a large message containing \( K \) short messages of the fine-grained program takes time \( \alpha_0 + K\alpha_1 \).

As in Section 3.1, we perform a complexity analysis for a pipe-line program of the form

\[
(d_{k+1}; \tau_k; d_k)^*
\]

As will be shown in Chapter 4, this structure is not changed under an \([M, K]\)-transformation. In the next sections, we discuss both the single process per processor case and the multiple processes per processor case.

One process per processor

When each processor accommodates one process, the sequence function of Example 2.22 can be applied, since the coarse-grained program has the same structure as the fine-grained one. A few substitutions are necessary: substitution of \( MK\beta \) for \( \beta \) (computation time), \( \alpha_0 + K\alpha_1 \) for \( \alpha \) (communication time), and \( N/M \) for \( N \) (number of processes in the program). This yields

\[
\sigma(d_k, i) = (\alpha_0 + K\alpha_1)(1 + \frac{N}{M}-k+2i) + MK\beta(\frac{N}{M}-k+i)
\]

for \( 0 \leq k < N/M \). Since messages contain \( K \) output values, the time for producing \( L \) output values is \( \sigma(d_k,(L/K)-1) \) (\( L \) is a multiple of \( K \)). For large \( L \), this results in a (parallel) execution time of \( \frac{LN\beta}{\frac{L}{K}(2(\alpha_0 + K\alpha_1) + MK\beta)} \). Thus, we obtain a speedup of

\[
\frac{LN\beta}{\frac{L}{K}(2(\alpha_0 + K\alpha_1) + MK\beta)} = \frac{N}{M} \frac{MK\beta}{2(\alpha_0 + K\alpha_1) + MK\beta}
\]

and an efficiency of

\[
\left(\frac{N}{M} \frac{MK\beta}{2(\alpha_0 + K\alpha_1) + MK\beta}\right) / \left(\frac{N}{M}\right) = \frac{1}{\frac{2(\alpha_0 + K\alpha_1)}{MK\beta} + 1}
\]
From the expression for efficiency, we infer that the communication overhead of the coarse-grained program is
\[
\frac{2(\alpha_0 + K\alpha_1)}{MK\beta}
\]
Hence, by applying an \([M, K]\)-transformation one can adjust the communication overhead and, thereby, the efficiency of the program by choosing appropriate values for parameters \(M\) and \(K\).

**Multiple processes per processor**

Next, we analyze the multiple processes per processor case. As in Section 3.1, \(P\) denotes the number of processors and \(q\) denotes the number of processes per processor. Assume that the total number of processes, \(N/M\), is a multiple of \(P\), then
\[
q = \frac{N}{MP}
\]

For communicating a message containing \(K\) values data processor DP needs \(\alpha_0\) time for communication setup and communication processor CP needs \(K\alpha_1\) time for the actual data transfer. Execution of a computation takes \(MK\beta\) time for the DP. From the sequence function we infer that one cycle in the program takes time \(2(\alpha_0 + K\alpha_1) + MK\beta\). The fraction of the cycle time the DP is occupied is
\[
f_d = \frac{2\alpha_0 + MK\beta}{2(\alpha_0 + K\alpha_1) + MK\beta}
\]
and the fraction of the cycle time the CP is occupied is
\[
f_c = \frac{2K\alpha_1}{2(\alpha_0 + K\alpha_1) + MK\beta}
\]

Note that again \(f_c + f_d = 1\). As a result, the case \(q*f_c < 1 \land q*f_d < 1\) yields \(q = 1\). We, therefore, consider the cases \(q*f_d \geq 1\) and \(q*f_c \geq 1\).

\(q*f_d \geq 1\)

Since \(f_c \leq f_d\), the DP is fully occupied and the CP is not. Since each of the \(q\) processes that are allocated to a processor execute \(L/K\) cycles, the parallel execution time equals
\[
t_p = q\frac{L}{K} (2\alpha_0 + MK\beta)
\]
resulting in a speedup of
\[
\frac{t_s}{t_p} = \frac{NL\beta}{qK(2\alpha_0 + MK\beta)} = \frac{N}{q} \frac{K\beta}{2\alpha_0 + MK\beta} = \frac{P}{2\alpha_0 + MK\beta} \frac{MK\beta}{MK\beta}
\]
and communication overhead
\[
\frac{2\alpha_0}{MK\beta}
\]
3.2. Communication Overhead

$q \times f_s \geq q \times f_d \geq 1$

Since $f_d \leq f_s$, the CP is fully occupied and the DP is not. For the parallel execution time we obtain

$$t_p = q \frac{L}{K} (2K \alpha_1) = 2qL\alpha_1$$

resulting in a speedup of

$$\frac{t_s}{t_p} = \frac{N\beta}{2qL\alpha_1} = \frac{N}{q} \frac{\beta}{2\alpha_1} = \frac{P_{M\beta}}{2\alpha_1}$$

Like in Section 3.1, the results of those two analyses can be combined, yielding an efficiency of

$$\min\left(\frac{MK\beta}{2\alpha_0 + MK\beta}, \frac{M\beta}{2\alpha_1}\right)$$

The efficiency of such a program can be adjusted by choosing appropriate values for parameters $M$ and $K$. For the computation of the optimal values for these parameters some boundary conditions have to be taken into account. One of these conditions is that each processor should have at least 2 processes: $q \geq 2$, which implies

$$M \leq \frac{N}{2P}$$

Furthermore, it is intuitively clear that for obtaining an optimal performance the CP should not be the bottleneck: $f_s \leq f_d$ and, hence,

$$2K\alpha_1 \leq 2\alpha_0 + MK\beta$$

This condition is certainly fulfilled if $M$ satisfies $M\beta \geq 2\alpha_1$. If both boundary conditions are satisfied, then the efficiency of the program equals

$$\frac{MK\beta}{2\alpha_0 + MK\beta} = \frac{1}{\frac{2\alpha_0}{MK\beta} + 1}$$

Hence, efficiency is optimal if $MK$ is as large as possible. Since $K$ is a divisor of $M$, the optimal choice for parameters $M$ and $K$ is $M$ as large as possible, given the boundary conditions, and $K = M$. 
Chapter 4

Deriving Programs of Parameterized Granularity

Methods for increasing the efficiency of parallel programs have been discussed in the previous chapter. One of these consists of a reduction of the communication overhead of programs by applying an $[M, K]$-transformation. Such a transformation leads to fewer communication statements, since a number of communications become internal and are replaced by assignment statements. Furthermore, communication time is reduced by composing a number of small messages into larger messages.

In this chapter, we present two examples of $[M, K]$-transformations. The starting point of such a transformation is a given fine-grained program. Throughout the construction of the coarse-grained program we can benefit from the knowledge gained in the construction of the original program, since this original program is obtained by substituting $(M, K) = (1, 1)$.

4.1 Pipe-line Computations

We consider a pipe-line program consisting of $N$ processes. Process $k$, $0 \leq k < N$, has an input channel $b_{k+1}$ and an output channel $b_k$ (cf. Figure 4.1) and it exhibits with respect to these channels the following communication behavior

$$(b_{k+1} ; b_k)$$

Let $E_k(i)$ be an expression such that $b_k(i) = E_k(i)$ is the i/o-relation of channel $b_k$. By applying an $[M, K]$-transformation, $M$ consecutive processes are combined into one large process and messages are composed of $K$ small messages of the original program (cf. Figure 4.2). In the coarse-grained program, the number of processes is reduced to $N/K$. Process $k$, $0 \leq k < N/K$, has output channel $b_k$. Since each communication along

$$b_k \rightarrow k \rightarrow b_{k+1}$$

Figure 4.1: (fine-grained) process of a pipe-line program
channel $\tilde{b}_h$ deals with $K$ messages of the original program, the i/o-relation for the $p$th component ($0 \leq p < K$) of channel $\tilde{b}_h$ is

$$ b_h(i)[p] = E_{hM}(iK + p) $$

for $i \geq 0$, since $\tilde{b}_h(i)[p] = b_{hM}(iK + p)$.

We now consider the communication behavior of a process of the coarse-grained program. Process $h$ is the composite of processes $hM$, $hM + 1$, ..., $hM + M - 1$. Considering the communication behavior of the fine-grained processes, we have

$$ b_{hM + M'i} < b_{hM + M'i + 1} < \ldots < b_{hM + Mi} $$

Hence, $b_{(h+1)M}(iK + p) < b_{hM}(iK + p)$ and for the channels of process $h$ of the coarse-grained program, we conclude that $b_{h+1}i < b_hi$. Since we aim at minimal buffering, the communication behavior is

$$ (\tilde{b}_{h+1} : \tilde{b}_h)^* $$

for $0 \leq h < \frac{N}{M}$. Notice that the communication behavior of the parameterized coarse-grained program is the same as the communication behavior of the fine-grained program.

**Example: the OCL-problem**

In this example, we show how to apply the $[M, K]$-transformation to a concrete pipeline program, viz. to the OCL-problem of Chapter 2. The corresponding fine-grained program consists of $N$ (the window length) processes. Each of these processes has three input and three output channels. The i/o-relations for channels $b_h$, $c_h$, and $d_h$ of process $k$ are (cf. Example 2.12)

$$ b_h(i) = (\# j : i - N + k < j \leq i : A(j) = A(i)) $$

$$ c_h(i) = A(i - N + k) $$

$$ d_h(i) = A(i) $$

In the coarse-grained program, we have channels $\tilde{b}_h$, $\tilde{c}_h$, and $\tilde{d}_h$ for process $h$, $0 \leq h < \frac{N}{M}$. From the i/o-relations of the fine-grained program and the relations given above (i.e. replace $i$ by $iK + p$ and $k$ by $hM$), it follows that the i/o-relations for these channels are

$$ \tilde{b}_h(i)[p] = (\# j : iK + p - N + hM < j \leq iK + p : A(j) = A(iK + p)) $$

$$ \tilde{c}_h(i)[p] = A(iK + p - N + hM) $$

$$ \tilde{d}_h(i)[p] = A(iK + p) $$
for \( i \geq 0 \) and \( 0 \leq p < K \).

The next step in the design of a program is to construct a set of equations that relate the values communicated along output channels to the values received by the input channels. To that end, we derive for \( \bar{b}_i[i][p] \), \( h \neq \frac{N}{M} - 1 \):

\[
\begin{align*}
\bar{b}_i[i][p] \\
&= \begin{cases} \\
\text{1/o-relation for } \bar{b}_i \\
\{ \# j : iK+p-N+hM < j \leq iK+p : A(j) = A(iK+p) \} \\
\{ \text{split off } M \text{ terms} \} \\
\{ \# j : iK+p-N+(h+1)M < j \leq iK+p : A(j) = A(iK+p) \} \\
+ \{ \# j : iK+p-N+hM < j \leq iK+p-N+(h+1)M : A(j) = A(iK+p) \} \\
\end{cases} \\
= \{ \text{1/o-relation for } \bar{b}_{h+1} \text{ and } \bar{d}_{h+1} \text{; range shift} \} \\
\bar{b}_{h+1}(i)[p] \\
+ \{ \# j : p < j \leq p+M : A(iK+j-N+hM) = \bar{d}_{h+1}(i)[p] \} \\
= \{ \text{split range} \} \\
\bar{b}_{h+1}(i)[p] \\
+ \{ \# j : p < j < M : A(iK+j-N+hM) = \bar{d}_{h+1}(i)[p] \} \\
+ \{ \# j : M \leq j \leq p+M : A(iK+j-N+hM) = \bar{d}_{h+1}(i)[p] \} \\
= \{ \text{1/o-relation } \bar{c}_{h+1} \} \\
\bar{b}_{h+1}(i)[p] \\
+ \{ \# j : p < j < M : A(iK+j-N+hM) = \bar{d}_{h+1}(i)[p] \} \\
+ \{ \# j : 0 \leq j \leq p : \bar{c}_{h+1}(i)[j] = \bar{d}_{h+1}(i)[p] \}
\end{align*}
\]

Furthermore, we have \( \bar{c}_h(i)[p] = A(iK+p-N+hM) \) and \( \bar{d}_h(i)[p] = \bar{d}_{h+1}(i)[p] \). In the computation of \( \bar{b}_i(i) \) and \( \bar{c}_i(i) \) the values \( A(iK+j-N+hM) \), for \( 0 \leq j < M \), are needed. Therefore, array \( V \) of dimension \( M \) is introduced that satisfies invariant

\[ R : V[j] = A(iK+j-N+hM) \]

for \( 0 \leq j < M \). Any initialization for array \( V \) will do, since for \( i=0 \) only negatively indexed elements of stream \( A \) are involved:

\[
\begin{align*}
j - N + hM \\
\leq & \{ \text{take } j=M-1 \text{ and } h=\frac{N}{M} - 1 \} \\
M - 1 - N + (\frac{N}{M} - 1)M \\
= & \{ \text{calculus} \} \\
-1
\end{align*}
\]

In order to establish \( R_{h+1}^i \) from \( R \), we derive

\[
A((i+1)K+j-N+hM) \\
= \{ \text{calculus} \}
\]
4.1. Pipe-line Computations

\[
\begin{align*}
\text{if} & \quad 0 \leq j < M - K & \rightarrow & \quad A(iK + (j+K) - N + hM) \\
\text{if} & \quad M - K \leq j < M & \rightarrow & \quad A(iK + (j+K-M) - N + (h+1)M) \\
\text{fi}
\end{align*}
\]

\[= \{ \text{R and i/o-relation } \bar{c}_{h+1} \} \]

\[
\text{if} \quad 0 \leq j < M - K \rightarrow V[j+K] \quad \text{or} \quad M - K \leq j < M \rightarrow \bar{c}_{h+1}(i)[j+K-M] \quad \text{fi}
\]

From this relation it may be observed that array \( V \) can easily be implemented as a cyclic array. After input \( \bar{c}_{h+1}(i) \) has been received, these \( K \) values are assigned to \( V \). The other \( M - K \) values are not affected.

The communication behavior of process \( h \) \((h \neq \frac{N}{M} - 1)\) of the coarse-grained program is

\[
(b_{h+1}, \bar{c}_{h+1}, d_{h+1} ; b_h, \bar{c}_h, d_h)
\]

resulting in a program for process \( h \) that has the following structure

```
initialize array V (optional)
; ( b_{h+1} ? vb, \bar{c}_{h+1} ? ve, d_{h+1} ? vd
    ; establish vb=b_h(i), vc=\bar{c}_h(i), and establish R_{h+1}
    ; b_{h+1} ? vb, \bar{c}_{h+1} ? ve, d_{h+1} ? vd )
```

Next, we address some performance-related issues for this instance of a pipe-line computation.

A comparison of the above program and the fine-grained program of Example 2.21 reveals that the number of computations between successive communications has increased. Since establishing \( vc=\bar{c}_h(i) \) as well as establishing \( R_{h+1} \) are rather simple computations, we consider the computation of \( b_h(i) \) only. For each of the \( K \) elements of this array one has to determine the multiplicity of a certain value in a bag of \( M \) values. An efficient implementation of this operation has a time complexity of \( O(\log M) \). Thus, the overall computation of the coarse-grained program has time complexity \( O(K \log M) \) for each step of the repetition.

By simply composing \( M \) processes of the fine-grained program, a solution in which the computation of each array element takes \( O(M) \) time is obtained, resulting in a time complexity of \( O(KM) \). Thus, designing a coarse-grained program opens the possibility for a more efficient implementation of some of the computations that have to be carried out. By applying an \([M, K]\)-transformation, parts of efficient sequential algorithms can be incorporated in the design.

Next, consider the data size of the program, which is an important issue for VLSI-implementations of programs. In the coarse-grained program an array \( V \) of dimension \( M \) and three arrays, \( vb, ve, \) and \( vd \), of dimension \( K \) have been introduced. Since the program consists of \( \frac{N}{M} \) processes, the total number of variables is \( \frac{N}{M} (M+3K) = N+3 \frac{M}{N} N \). The fine-grained program of Example 2.21 has 4 variables for each process, yielding a total of \( 4N \) variables for the overall program. Since \( K \leq M \), this example shows that applying an \([M, K]\)-transformation may reduce the data size of a program.
4.2 Stack-like Computations

In this section, we consider a stack-like computation that consists of an unbounded number of processes, like the Paliindrome Recognizer. Process \(k\), 0 ≤ \(k\), has two input channels, denoted by \(a_k\) and \(b_{k+1}\), and two output channels, denoted by \(b_k\) and \(b_{k+1}\) (cf. Figure 4.3), and it has communication behavior

\[ a_k; (b_k; a_k)^{L-1}; (b_k, a_{k+1}; a_k, b_{k+1})^* \]

for some positive constant \(L\).

Assume that, like in the pipe-line program, channels \(a_k\) and \(b_k\) satisfy i/o-relations \(a_k(i) = F_k(i)\) and \(b_k(i) = G_k(i)\). By applying an \([M,K]\)-transformation (see Figure 4.4) output channels \(\bar{a}_k\) and \(\bar{b}_k\) of process \(h\) of the coarse-grained program should satisfy the following i/o-relations

\[ \bar{a}_k(i)[p] = F_{hM}(iK+p) \]
\[ \bar{b}_k(i)[p] = G_{hM}(iK+p) \]

for \(i \geq 0\) and \(0 \leq p < K\).

Next, we consider the communication behavior of the processes of the coarse-grained program. Assume that in the fine-grained processes each output value that is computed depends on all proceeding input values received, except for the output values of channel \(a_{k+1}\); these only depend on channel \(a_k\). For fine-grained process \(k\) the following partial order on communication actions holds

\[ a_k[i] < b_{k}[i] \]
\[ b_{k+1}[i] < b_{k}[i+L] \]
\[ a_k[i+L-1] < a_{k+1}[i] \]

Course-grained process \(h\) is the composite of processes \(hM, hM+1, \ldots, hM+M-1\). We consider the partial orders on the communications along channels \(\bar{a}_k\) and \(\bar{b}_k\) first. From \(a_{hM}(iK+p) < b_{hM}(iK+p)\) we infer

\[ \bar{a}_k[i] < \bar{b}_k[i] \]
4.2. Stack-like Computations

Since $b_{hM+m+1}^{i}(iK+p) < b_{hM+m+1}^{i}(iK+p+L) \ (0 \leq m < M)$, for channels $\tilde{b}_{h}$ and $\tilde{b}_{h+1}$ we have

$$\tilde{b}_{h+1}^{(i+LH)} < \tilde{b}_{h}^{(i)}$$

where $H$ is defined by $H = M/K$. By using $a_{hM+m+1}^{i}(iK+p+L-1) < a_{hM+m+1}^{i}(iK+p)$, we finally obtain

$$\tilde{a}_{h}^{(i+(L-1)H)} < \tilde{a}_{h+1}^{(i)}$$

for channels $\tilde{a}_{h}$ and $\tilde{a}_{h+1}$.

**Remark 4.1**

In case the output values of channel $a_{h+1}$ also depend on input values received along channel $b_{h+1}$, i.e., if $b_{h+1}^{(i)} < a_{h+1}^{(i+1)}$, we can derive a contradiction for $K \neq 1$: for $k = hM + M - 1$, this partial order yields $b_{h+1}^{(i+1)}(iK+p) < a_{h+1}^{(i+1)}(iK+p+1)$. Hence, for $0 \leq p < K - 1$, we obtain $\tilde{a}_{h+1}^{(i)} < \tilde{a}_{h+1}^{(i)}$, which is in contradiction to $\tilde{a}_{h+1}^{(i)} < \tilde{a}_{h+1}^{(i+1)}$ derived above.

(Notice that for $p = K - 1$ we have $\tilde{b}_{h+1}^{(i)} < \tilde{a}_{h+1}^{(i+1)}$. Therefore, $K = 1$ gives no contradiction as can be deduced from the overall communication behavior derived in the sequel.)

□

From the first two equations the communication behavior of process $h$ with respect to channel $\tilde{b}_{h}$ follows:

$${\text{CB}}(\tilde{b}_{h}) : \tilde{a}_{h}^{(i)}(\tilde{b}_{h}; \tilde{a}_{h})^{LH-1} ; (\tilde{b}_{h}; \tilde{a}_{h}, \tilde{b}_{h+1})^{*}$$

And from the third equation we infer

$${\text{CB}}(\tilde{a}_{h+1}) : \tilde{a}_{h}^{(i-LH+1)} ; (\tilde{a}_{h+1}; \tilde{a}_{h})^{*}$$

Combining these communication behaviors in an overall communication behavior for process $h$ yields

$${\text{CB}}_{h} : \tilde{a}_{h}^{(i)}(\tilde{b}_{h}; \tilde{a}_{h})^{LH-H} ; (\tilde{b}_{h}, \tilde{a}_{h+1}; \tilde{a}_{h})^{H-1} ; (\tilde{b}_{h}, \tilde{a}_{h+1}; \tilde{a}_{h}, \tilde{b}_{h+1})^{*}$$

Since $C_{B_{h}}[\{\tilde{a}_{h+1}, \tilde{b}_{h+1}\}] \neq C_{B_{h}}[\{\tilde{a}_{h+1}, \tilde{b}_{h+1}\}]$, we adjust communication behavior $C_{B_{h}}$ by delaying communications along channel $\tilde{a}_{h+1}$ resulting in

$${\text{CB}}_{h}' : \tilde{a}_{h}^{(i)}(\tilde{b}_{h}; \tilde{a}_{h})^{LH-1} ; (\tilde{b}_{h}, \tilde{a}_{h+1}; \tilde{a}_{h}, \tilde{b}_{h+1})^{*}$$

Hence, communications along channel $\tilde{a}_{h}$ are buffered.

**Example: the palindrome recognizer**

As a more concrete example we apply the $[M, K]$-transformation to the Palindrome Recognizer of Chapter 2. The fine-grained program consists of an unbounded array of processes. Each process $h, h \geq 0$, has two input and two output channels. Channels $a_{h}$ and $b_{h}$ satisfy i/o-relations (cf. Example 2.10)

$$a_{h}(i) = A(i+k)$$

$$b_{h}(i) \equiv (\forall j, h : 0 \leq j \wedge 0 \leq h \wedge j+h = i : A(j+k) = A(h+k))$$
In the coarse-grained program the corresponding channels are \( \overline{a}_h \) and \( \overline{b}_h \) \((h \geq 0)\). From the \( i/o \)-relations of the fine-grained program it follows that the \( i/o \)-relations for these channels are (replace \( k \) and \( i \) by \( hM \) and \( iK+p \), respectively):

\[
\begin{align*}
a_h(i)[p] &= A(iK+p+hM) \\
\overline{b}_h(i)[p] &= (\forall j, k : 0 \leq j \land 0 \leq k \land j+k=iK+p : A(j+hM) = A(k+hM))
\end{align*}
\]

From the communication behavior of the fine-grained program (cf. Example 2.17), i.e.

\[
a_b; (b_1; a_k; (b_2, a_{k+1}; a_k, b_{k+1}))^*
\]

we observe that we can apply the general scheme above for \( L=2 \). Hence, the coarse-grained program has as communication behavior

\[
\overline{a}_h; (\overline{b}_h; a_k)^{2H-1}; (\overline{b}_h, \overline{a}_{h+1}; \overline{a}_h, \overline{b}_{h+1})^*
\]

where \( H=\frac{M}{K} \).

For the values communicated along channel \( \overline{b}_h \) we derive \((i \geq 2H)\)

\[
\begin{align*}
\overline{b}_h(i)[p] &= (\text{i/o-relation } \overline{b}_h) \\
&= (\forall j, k : 0 \leq j \land 0 \leq k \land j+k=iK+p : A(j+hM) = A(k+hM)) \\
&= (\text{split off } M \text{ terms}) \\
&= (\forall j, k : M \leq j \land M \leq k \land j+k=iK+p : A(j+hM) = A(k+hM)) \\
&\quad \land (\forall j, k : 0 \leq j < M \lor 0 \leq k < M \land j+k=iK+p \\
&\quad \quad : A(j+hM) = A(k+hM)) \\
&= (\text{range shift; } M = HK; \text{ symmetry}) \\
&= (\forall j, k : 0 \leq j \land 0 \leq k \land j+k=(i-2H)K+p : A(j+(h+1)M) = A((k+(h+1)M)) \\
&\quad \land (\forall j, k : 0 \leq j < M \land j+k=iK+p : A(j+hM) = A(k+hM)) \\
&= (\text{i/o-relation } \overline{b}_{h+1}) \\
&= (\forall j, k : 0 \leq j < M \land j+k=iK+p : A(j+hM) = A(k+hM))
\end{align*}
\]

From the expressions for the values to be communicated along channel \( \overline{b}_h \), the elements from input stream \( A \) that are required for these values can now be determined.

For the computation of \( b_h(i), 0 \leq i < 2H \), the elements \( A(j+hM) \) for \( 0 \leq j < (i+1)K \) are needed. For the computation of \( b_h(i), i \geq 2H \), the elements \( A(j+hM) \) for \( 0 \leq j < M \) and the elements \( A(iK-M+k+hM) \) for \( 0 < k < M+K \). Since \( \overline{a}_h[i][p]=A(iK+p+hM) \), we introduce auxiliary arrays \( U \) and \( V \) satisfying

\[
\begin{align*}
R_0 : &\quad (\forall j : 0 \leq j < iK : U[j] = A(j+hM)) \\
R_1 : &\quad (\forall j : 0 < j < M : V[j] = A(iK-M+j+hM))
\end{align*}
\]
4.2. Stack-like Computations

This leads to the following structure for the course-grained program of process \( h \) of the palindrome recognizer:

\[
\begin{align*}
&\text{\( \tilde{a}_h\)?u} \\
&\quad; \begin{cases} \{ R_0 \} \\
&\quad; \text{establish } v_b = \tilde{b}_h(i) \text{ and } (R_0)_{i+1}^j \\
&\quad; \tilde{b}_h[v_b \\
&\quad; \tilde{a}_h?va \\
&\quad; j \in (R_0)_{i+1}^j \\
&\quad; \{ (R_0)_{i+1}^j, \text{ hence } U[j] = A(j+hM) \text{ for } 0 \leq j < M \} \\
&\quad; \text{establish } (R_1)_{i+1}^j \\
&\quad; \begin{cases} \{ R_1 \} \\
&\quad; \text{establish } v_b = \tilde{b}_h(i) \text{ and } (R_1)_{i+1}^j \\
&\quad; \tilde{b}_h[v_b \\
&\quad; \tilde{a}_h?va \\
&\quad; j \in (R_1)_{i+1}^j \\
&\quad; \begin{cases} \{ R_1 \} \\
&\quad; \text{establish } v_b = \tilde{b}_h(i) \text{ and } v_a = \tilde{a}_{h+1}(i-2H+1) \text{ and } (R_1)_{i+1}^j \\
&\quad; \tilde{b}_h[v_b, \tilde{a}_{h+1}?va \\
&\quad; \tilde{a}_h?va, \tilde{b}_{h+1}?vb \\
&\quad; j \in (R_1)_{i+1}^j \\
&\quad; \end{cases} \\
&\quad; \end{cases}
\end{cases}
\end{align*}
\]

Since \( i=0 \) implies that the qualification in \( R_0 \) has an empty range, initialization of invariant \( R_0 \) is trivial. For establishing \( (R_0)_{i+1}^j \), only indices \( j \) with \( iK \leq j < (i+1)K \) have to be considered. We derive

\[
\begin{align*}
A(j+hM) & = \{ \text{calculus} \} \\
& = \{ \text{i/o-relation } \tilde{a}_h \} \\
& = \tilde{a}_h(i)[j-iK]
\end{align*}
\]

Establishing \( R_1 \) turns out to be simple, since \( i=H \) yields \( A(HK-M+j+hM) = A(j+hM) \) which equals \( U[j] \) on account of \( (R_0)_{i+1}^j \). Thus, \( V[j] = U[j] \) for \( 0 \leq j < M \) implies \( (R_1)_{i+1}^j \).

**Remark 4.2**

As a matter of fact, the initialisation of invariant \( R_1 \) may be omitted, since for \( H \leq i < 2H-1 \), invariant \( R_1 \) may be replaced by \( R'_1 \) satisfying

\[
(\forall j: 2M-iK < j < M : V[j] = A(iK-M+j+hM))
\]

Notice that \( (R'_1)_{i+1}^j \equiv \text{true} \) and \( (R'_1)_{i+1}^j \equiv R_1 \). \( \square \)
For \((R_t)_{i+1}\), we derive

\[
A((i+1)K - M + j + hM)
\]

\{ calculus \}

\[
A(iK - M + (j+K) + hM)
\]

\{ \(R_t\) \}

\[
\text{if } j+K < M \rightarrow V[j+K][j+K \geq M \rightarrow A(iK + (j+K-M) + hM) \text{fi}
\]

\{ i/o-relation \(\tilde{a}_h\) \}

\[
\text{if } j < M - K \rightarrow V[j+K][j \geq M - K \rightarrow a_h(i)[j+K-M] \text{fi}
\]

From this expression it follows that array \(V\) can be implemented as a cyclic array, similar to array \(V\) in the coarse-grained program for the OCL-problem. Finally, for establishing \(va = \tilde{a}_{h+1}(i-2H+1)\) we derive

\[
\tilde{a}_{h+1}(i-2H+1)[p]
\]

\{ i/o-relation \(\tilde{a}_{h+1}\) \}

\[
A((i-2H+1)K + p + (h+1)M)
\]

\{ M = HK \}

\[
A((i+1)K - 2M + p + hM + M)
\]

\{ calculus \}

\[
A((i+1)K - M + p + hM)
\]

\{ \((R_t)_{i+1}\) \}

\[
V[p]
\]

Establishing \(vb = \tilde{b}_h(i)\) is the main computation in each step of the repetition of the program. In such a step, process \(b\) computes for each \(p\), \(0 \leq p < K\), a statement of the form:

\[
\text{vb} [p] := \text{vb} [p] \wedge (\forall j : 0 \leq j < M \wedge \ldots : A(\ldots) = A(\ldots))
\]

In the worst case, all \(M\) equalities of this conjunction have to be evaluated. In an efficient implementation of this computation, no more equality is evaluated once a single inequality has been found. Moreover, if \(\neg \text{vb} [p]\) holds as a precondition, this conjunction need not be considered at all, whereas in the fine-grained program for Palindrome Recognition each equality is evaluated. Thus, only in the worst case (i.e. if \(\text{vb} [p]\) holds and stream \(A\) is constant) does the total number of computations of the coarse-grained program equals that in the fine-grained program. On the average, however, the coarse-grained program is more efficient.

With respect to the data size of the coarse-grained program, we observe that the program contains two (basic) types of variables, viz. symbols and booleans. Each process of the coarse-grained program has \(2M + K - 1\) variables of type symbol (arrays \(U, V\), and \(va\)) and \(K\) variables of type boolean (array \(vb\)). \(M\) processes of the fine-grained program contain a total number of \(2M\) variables of type symbol and \(M\)
variables of type boolean. If the ratio between the cost of a symbol and a boolean is $n$, one can easily derive that the coarse-grained program has a smaller data size than the fine-grained program for $M > (n+1)K - n$. 
Chapter 5

General Scheme

In the previous chapter, an \([M, K]\)-transformation has been applied to two example programs: a pipe-line computation and a stack-like computation. In both examples, the set of i/o-relations of the fine-grained parallel programs was adjusted into a new set of i/o-relations that met the requirements of the specified \([M, K]\)-transformation. From this new set of i/o-relations a course-grained program was obtained.

In this chapter, we address a more general question: given a fine-grained program, is it possible obtain a course-grained program by applying an \([M, K]\)-transformation? For both types of computations that we consider, the answer turns out to be affirmative. We show a method to construct a coarse-grained program starting from the set of equations that determines how the values communicated along the input and output channels of the fine-grained processes are related.

Although the method of this chapter can be applied to obtain coarse-grained programs, it is in general recommended to design coarse-grained programs by using the approach given in the previous chapter. As stated before, the method presented in this chapter is quite general and only needs the set of equations that prescribes the values communicated along channels as a starting point. In this way, a coarse-grained program is obtained that, in a sense, simulates the fine-grained parallel program. The general applicability of the method does not allow the inclusion of problem-specific information in the design of the program, as would have been possible by designing a coarse-grained program starting from the set of i/o-relations.

5.1 Pipe-line Computations

In this section we consider pipe-line computations. Let processes \(k\) of such a program have input channel \(b_{k+1}\) and output channel \(b_{k}\) (cf. Figure 5.1). Process \(k\) exhibits

\[
\begin{array}{c}
\text{input} \\
\text{output}
\end{array}
\]

Figure 5.1: fine-grained and coarse-grained process of a pipe-line computation
5.1. Pipe-line Computations

communication behavior

\[(b_{k+1}; b_k)^*\]

and contains a local variable, denoted by \(w_k\). This local variable is updated in each step of the repetition by applying a function that has two arguments, viz. the current value of the local variable and the input value received along channel \(b_{k+1}\). The output value to be communicated along channel \(b_k\) is also computed from these two values. We then have the following set of equations that output channel \(b_k\) and local variable \(w_k\) satisfy

\[
b_k(i) = F_k(b_{k+1}(i), w_k(i))
\]
\[
w_k(i+1) = T_k(b_{k+1}(i), w_k(i))
\]

From the communication behavior and this set of equations, the program text for process \(k\) is easily deduced:

```plaintext
{ u : T_0; w : T_1;
  initialize w = w_k(0)
  ; ( b_{k+1}? u
    {u = b_{k+1}(i) \land w = w_k(i)}
    ; u, w := F_k(u, w), T_k(u, w)
    {u = b_k(i) \land w = w_k(i+1)}
    ; b_k! u
  )
}
```

**Example 5.1** (Functions \(F_k\) and \(T_k\) for the OCL-problem)

In this example we show how functions \(F_k\) and \(T_k\), as presented above, can be constructed. We use as example the OCL-problem.

Examples 2.12 and 2.21 show that each fine-grained process in the program for the OCL-problem has three input and three output channels and one local variable. The set of defining equations of the fine-grained program is (channels from the original fine-grained program are augmented with "" to avoid name-clashes with channel names of the general framework)

\[
b_k(i) = b_{k+1}(i) + [c_{k+1}(i) = d_{k+1}(i)]
\]
\[
c_k(0) = 0
\]
\[
c_k(i+1) = c_{k+1}(i)
\]
\[
d_k(i) = d_{k+1}(i)
\]

Channel \(b_k\) and local variable \(w_k\) of the general framework are now defined as

\[
b_k(i) = (b_k(i), c_k(i), d_k(i))
\]
\[
w_k(i) = c_k(i+1)
\]

From these relations, functions \(F_k\) and \(T_k\) are readily obtained:

\[
F_k((x_0, x_1, x_2), y) = (x_0 + [x_1 = x_2], y, x_2)
\]
\[
T_k((x_0, x_1, x_2), y) = x_2
\]
Note that in this example both \( F_k \) and \( T_k \) are independent of \( k \).

We now show how to construct a coarse-grained program from the above fine-grained version by applying an \([M,K]\)-transformation. Consider a coarse-grained process \( h \). As shown in Section 4.1, the communication behavior of the coarse-grained program reflects that of the fine-grained program:

\[(b_{h+1}, b_h)\]^*

and for the values communicated along channel \( b_h \) we have

\[b_h(i)[p] = b_{hM}(iK+p)\]

for \( i \geq 0 \) and \( 0 \leq p < K \).

We prove that a coarse-grained program can be constructed from a fine-grained program, irrespective of functions \( F_k \) and \( T_k \), by deriving statement \( S \) for the coarse-grained program such that

\[
\begin{align*}
\text{\texttt{\{} } & \text{\texttt{\{} u \texttt{\}}} : \text{array \([0..K]\)} \text{ of } T_0; \\
& w : \text{array \([0..M]\)} \text{ of } T_1; \\
& \text{initialize } R_0 \\
& b_{h+1} : \text{channel } b_h \\
& \{ \bar{u} = b_{h+1}(i) \land R \} \\
& S \\
& \{ u = b_h(i) \land R^c \} \\
& b_h \bar{u} \\
\text{\texttt{\}}}^*
\end{align*}
\]

where \( R \) is defined as

\[R : w[p] = w_{hM+p}(iK)\]

for \( 0 \leq p < M \).

Notational convention: We adopt the following notational convention for arrays and array elements: \( \bar{w} \) denotes an (entire) array and the \( i \)-th element of this array is denoted by \( w[i] \).

Statement \( S \)

Statement \( S \) of the above program program is specified by

\[
\begin{align*}
\{ I : & (\forall p : 0 \leq p < K : w[p] = b_{hM+p}(iK+p)) \\
& \land (\forall p : 0 \leq p < M : w[p] = w_{hM+p}(iK)) \\
& J : & (\forall p : 0 \leq p < K : u[p] = b_{hM}(iK+p)) \\
& \land (\forall p : 0 \leq p < M : w[p] = w_{hM+p}((i+1)K)) \\
& S \\
\}
\end{align*}
\]
To establish postcondition \( J \) from precondition \( I \), variable \( m \) and accompanying invariant \( P \), the conjunction of \( P_0 \) and \( P_1 \), are introduced:

\[
P_0: \quad u[p] = b_{k M + m}(iK + p) \quad \text{for } 0 \leq p < K
\]

\[
P_1: \quad u[p] = \begin{cases} 
u_{h M + p}(iK) & \text{for } 0 \leq p < m \\ \nu_{h M + p}((i+1)K) & \text{for } m \leq p < M \end{cases}
\]

Invariant \( P \) is established by \( m := M \). Furthermore, \( P_0^m \) implies postcondition \( J \). Establishing \( P \) from \( P_{m+1}^m \) requires a repetition with invariant \( Q \) and the introduction of variable \( n \):

\[
Q_0: \quad u[p] = \begin{cases} b_{h M + m}(iK + p) & \text{for } 0 \leq p < n \\ b_{h M + m+1}(iK + p) & \text{for } n \leq p < K \end{cases}
\]

\[
Q_1: \quad w[m] = w_{h M + m}(iK + n)
\]

Invariant \( Q \) is established by \( n := 0 \), using \( P_{m+1}^m \), and invariant \( P \) is implied by \( Q_{n+1}^n \). For establishing \( Q_{n+1}^n \) from \( Q \), we derive

\[
b_{h M + m}(iK + n)
= \{ \text{defining equation for } b \}
\]

\[
F_{h M + m}(b_{h M + m+1}(iK + n), w_{h M + m}(iK + n))
= \{ Q \}
\]

\[
F_{h M + m}(u[n], w[m])
\]

and

\[
w_{h M + m}(iK + n+1)
= \{ \text{defining equation for } w \}
\]

\[
T_{h M + m}(b_{h M + m+1}(iK + n), w_{h M + m}(iK + n))
= \{ Q \}
\]

\[
T_{h M + m}(u[n], w[m])
\]

As a result, we obtain the following program text for statement \( S \) (where \( f_b \) is a shorthand for \( F_{h M + b} \))

\[
\{ I \}
\]

\[
m := M \quad \{ P \}
\]

\[
; \text{do } m \neq 0 \rightarrow \\
\qquad m, n := m - 1, 0 \quad \{ P^m_{m+1} \land Q \}
\]

\[
; \text{do } n \neq K \rightarrow \\
\qquad u[n], w[m] := f_m(u[n], w[m]), t_m(u[n], w[m]) \quad \{ Q^n_{n+1} \}
\]

\[
; \quad n := n + 1
\]

\[
; \text{od} \quad \{ P \}
\]

\[
; \text{od} \quad \{ P_0^m \}
\]

\[
\{ J \}
\]
Figure 5.2: fine-grained and coarse-grained process of a stack-like computation

**Remark 5.2** (Substitution of $M=1$ and $K=1$)

By substituting $M=1$ and $K=1$ in the coarse-grained program one should obtain the fine-grained program. In this case statement $\mathcal{S}$ can be simplified to

$$u[0], w[0] := f_0(u[0],w[0]), t_0(u[0],w[0])$$

which is indeed the statement that is performed in each step of the repetition of the fine-grained program.

Finally, we address the initialization of $w$. From the definition of $R$ we infer

$$(R_0)^0 : w[p] = w_{M+1}(0)$$

for $0 \leq p < M$. In the fine-grained program the initialization of the repetition consists of establishing $w = w_k(0)$. This computation is performed without using any input values received by the process. Therefore, establishing $(R_0)^0$ can also be performed without using input values.

### 5.2 Stack-like Computations

As a second type of computation we consider stack-like computations with an unbounded number of processes. Let process $k$ ($k \geq 0$) have two input channels, $a_k$ and $b_{k+1}$, and two output channels, $b_k$ and $a_{k+1}$ (cf. Figure 5.2). Process $k$ exhibits communication behavior

$$a_k; (b_k; a_k)^{L-1}; (b_k, a_{k+1} ; a_k, b_{k+1})^*$$

where $L$ is a constant ($L \geq 1$).

In Section 4.2, it is argued that output values communicated along channel $a_{k+1}$ depend on input values received along channel $a_k$, whereas output values of channel $b_k$ may depend on all input values received. It is for this reason that we introduce two local variables, instead of one, per process. These variables are denoted by $v_k$ and $w_k$, where the value of $v_k$ depends on the input received along channel $a_k$ only. The following set of equations is satisfied by the process

$$b_k(i) = F_k(a_k(i), b_{k+1}(i-L), v_k(i), w_k(i))$$

$$a_{k+1}(i) = G_k(a_k(i+L-1), v_k(i+L-1))$$

$$v_k(i+1) = S_k(a_k(i), v_k(i))$$

$$w_k(i+1) = T_k(a_k(i), b_{k+1}(i-L), v_k(i), w_k(i))$$
5.2. Stack-like Computations

From this set of equations and the communication behavior we obtain the following program text for the fine-grained program:

\[
[ \begin{align*}
&x : T_0; u : T_1; v : T_2; w : T_3; \\
&\text{initialize } v = v_k(0) \text{ and } w = w_k(0) \\
&; a_k ? x \\
&; ( \{ x = a_k(i) \land v = v_k(i) \land w = w_k(i) \} \\
&; u, v, w := F_k(x, u, v, w), S_k(x, v), T_k(x, u, v, w) \\
&; \{ u = b_k(i) \land v = v_k(i+1) \land w = w_k(i+1) \} \\
&; b_k ! u \\
&; a_k ? x \\
&; ( ( \{ x = b_k(i) \land u = b_{k+1}(i-L) \land v = v_k(i) \land w = w_k(i) \} \\
&; x, u, v, w := G_k(x, v), F_k(x, u, v, w), S_k(x, v), T_k(x, u, v, w) \\
&; \{ x = a_{k+1}(i-L+1) \land u = b_k(i) \land v = v_k(i+1) \land w = w_k(i+1) \} \\
&; b_k ! u, a_{k+1} ! x \\
&; a_k ? x, b_{k+1} ? u \\
&; ) \\
&] \\
\]

Example 5.3 (Functions $F_k$, $G_k$, and $T_k$ for the Palindrome Recognizer)
In this example we show how functions $F_k$, $G_k$, and $T_k$ can be constructed for a particular instance of a stack-like computation, viz. the Palindrome Recognizer.

From Examples 2.10 it follows that values received along the input channel of process $k$ are propagated directly to the input channel of process $k+1$. On account of this observation, local variable $v_k$ can be omitted. The defining equations of process $k$ of the Palindrome Recognizer are (channels from the original fine-grained program are augmented with "!"):

\[
\begin{align*}
\hat{b}_k(0) &= \text{true} \\
\hat{b}_k(1) &= (\hat{a}_k(0) = \hat{a}_k(1)) \\
\hat{b}_k(i+2) &= (\hat{a}_k(0) = \hat{a}_k(i+2)) \land \hat{b}_{k+1}(i) \\
\hat{a}_{k+1}(i) &= \hat{a}_k(i+1)
\end{align*}
\]

A local variable is introduced for index $i$ to determine if the defining equation for $\hat{b}_k(0)$, $\hat{b}_k(1)$, or $\hat{b}_k(i+2)$ should be applied. Now, channels $a_k$ and $b_k$ and local variable $w_k$ of the general framework are defined as

\[
\begin{align*}
\hat{b}_k(i) &= \hat{b}_k(i) \\
\hat{a}_k(i) &= \hat{a}_k(i) \\
\hat{w}_k(i) &= (i, \hat{a}_k(0))
\end{align*}
\]
From these relations, functions $F_k$, $G_k$, and $T_k$ are readily derived

$$F_k(x, y, (x_0, z_1)) = \begin{cases} \text{true} & \text{if } z_0 = 0 \\ (x = z_1) & \text{if } z_0 = 1 \\ (x = z_1) \land y & \text{if } z_0 \geq 2 \end{cases}$$

$$G_k(x) = x$$

$$T_k(x, y, (x_0, z_1)) = \begin{cases} (1, x) & \text{if } z_0 = 0 \\ (x_0 + 1, z_1) & \text{if } z_0 \geq 1 \end{cases}$$

Note that both $F_k$ and $G_k$ as well as $T_k$ are independent of $k$.

The construction of a coarse-grained program for stack-like programs is presented in Appendix A. We conclude this section with an example in which variable $v_k$ of the general scheme cannot be omitted. As explained in Appendix A, this gives rise to the introduction of an additional array of dimension $K$ in the program.

**Example 5.4 (Functions $S_k$ and $G_k$ for Dirichlet convolution [27])**

In [27] the derivation of a fine-grained parallel program for Dirichlet convolution is presented. In this example, we only consider input channels $e_k$ and $f_k$ of process $k$ that satisfy the following set of equations (see also Example 2.14)

$$e_{k+1}(i) = \begin{cases} e_k(i) & \text{if } -(i+1 \mid k+1) \lor i = 0 \\ e_k(i-1) & \text{if } (i+1 \mid k+1) \land i^2 > i+k \\ f_k(i-1) & \text{if } (i+1 \mid k+1) \land 0 < i^2 \leq i+k \end{cases}$$

$$f_{k+1}(i) = f_k(i)$$

In this example it suffices to abbreviate the equation for output channel $e_{k+1}$ to $e_{k+1}(i) = \text{CASE}(e_k(i), e_k(i-1), f_k(i-1))$. The set of equations yields the following definitions of channel $a_k$ and local variable $v_k$ of the general framework

$$a_k(i) = (e_k(i), f_k(i))$$

$$v_k(i) = (e_k(i-1), f_k(i-1))$$

From $v_k(i+1) = S_k(a_k(i), v_k(i))$ it follows that

$$S_k((x_0, z_1), (y_0, y_1)) = (x_0, z_1)$$

on account of $v_k(i+1) = a_k(i)$. For function $G_k$ defined by $a_{k+1}(i) = G_k(a_k(i), v_k(i))$ we derive

$$a_{k+1}(i)$$

$$= \begin{cases} \text{definition } a_k(i) \} \\ (e_{k+1}(i), f_{k+1}(i)) \} \\ \text{defining equations for } e_k \text{ and } f_k \} \\ (\text{CASE}(e_k(i), e_k(i-1), f_k(i-1)), f_k(i)) \} \\ \{ \text{let } a_k(i) = (x_0, z_1) \text{ and } v_k(i) = (y_0, y_1) \}$$
5.2. Stack-like Computations

\((\text{CASE}(x_0, y_0, y_1), x_1)\)

We conclude that \(G_{\#}((x_0, x_1), (y_0, y_1)) = (\text{CASE}(x_0, y_0, y_1), x_1)\).

From the general scheme in Appendix A, we infer that in each process of the coarse-grained program channel \(e_k\) gives rise to \(2(M+K)\) variables and variable \(v_k\) gives rise to \(2(M+K)\) variables. Hence, the total number of variables needed for channels \(e_k\) and \(f_k\) of fine-grained program is \(4M+4K\).

\(\Box\)

In this example, variable \(v_k\) cannot be omitted. In Appendix A it is explained that on account of this observation an additional array of dimension \(K\) has to be introduced and, moreover, for each process an additional computation of \(O(K^2)\) is needed to produce the input stream for the neighbor process. A drawback of this general scheme is that application specific information is not used in the design of the parallel program. This may lead to programs that do not have optimal performance. An example thereof is the program for Dirichlet convolution presented above. In the approach of the previous chapter, i.e. start the design with i/o-relations instead of the set of equations, it is possible to exploit properties of the problem. As an illustration, we apply the method of Chapter 4 to the problem of computing the Dirichlet convolution.

**Example 5.5** (design of a coarse-grained program for Dirichlet convolution)

In this example, we discuss a part of the design of a coarse-grained parallel program for Dirichlet convolution [27]. We consider input channels \(e_k\) and \(f_k\) of process \(k\). The i/o-relations for these channels is

\[
e_k(i) = F(1 + j \text{ div } (i+1))
\]

\[
f_k(i) = F(1 + i)
\]

When applying an \([M,K]-\)transformation, the following i/o-relations for channels \(\bar{e}_k\) and \(\bar{f}_k\) of the coarse-grained program are obtained

\[
\bar{e}_k(i)[p] = F(1 + (hM) \text{ div } (iK+p+1))
\]

\[
\bar{f}_k(i)[p] = F(1 + iK + p)
\]

for \(i \geq 0\) and \(0 \leq p < K\). At this stage in the design we typically address the construction of a statement-list \(S\) that is specified as follows

\[
\{(\bar{z} = \bar{e}_k(i) \land \bar{y} = \bar{f}_k(i) \land R)\}
\]

\(S\)

\[
\{(\bar{z} = \bar{e}_{k+1}(i) \land \bar{y} = \bar{f}_{k+1}(i) \land R_{k+1})\}
\]

Since \(\bar{f}_{k+1}(i) = \bar{f}_k(i)\), we focus our attention on the computation of \(\bar{e}_{k+1}(i)\) from \(\bar{e}_k(i)\).

For this computation \(M+1\) additional variables are needed:

\[
R_0 : (\forall p : 0 \leq p < M : xz[p] = F(1 + (hM+p) \text{ div } (iK)))
\]

\[
R_1 : yy = F(1 + iK - 1)
\]
Since $F(1+(i+1)K-1) = F(1+iK+(K-1))$, $(R_1)_{i+1}$ is established by $y_2 := y[K-1]$.

The computation consists of a repetition with invariant $P$

\[ P_0: \quad z[p] = F(1 + (hM+m) \div (iK+p+1)) \quad \text{for } 0 \leq p < K \]
\[ P_1: \quad x_2[p] = \begin{cases} 
F(1 + (hM+p) \div ((i+1)K)) & \text{for } 0 \leq p < m \\
F(1 + (hM+p) \div (iK)) & \text{for } m \leq p < M
\end{cases} \]

Initially, $P$ is satisfied by $m := 0$ and the postcondition is implied by $P_0^m$. In order to establish $P_{m+1}^m$ from $P$, we derive for $P_{m+1}^m$

\[ F(1 + (hM+m+1) \div (iK+p+1)) = \begin{cases} 
\text{definition of CASE, see Example 5.4} \\
\text{CASE}(F(1 + (hM+m) \div (iK+p+1)), F(1 + (hM+m) \div (iK+p)) \\
\quad F(1 + (iK+p-1))
\end{cases} \]
\[ \begin{cases} 
\text{distinguish cases } p = 0 \text{ and } p \neq 0; \text{ invariant } P \\
\text{if } p = 0 \rightarrow \text{CASE}(x[0], x[x[m]], y_2) \\
\quad p \neq 0 \rightarrow \text{CASE}(x[p], x[p-1], y[p-1])
\end{cases} \]

and for $(P_1)_{m+1}^m$

\[ F(1 + (hM+m) \div ((i+1)K))) = \begin{cases} 
\text{calculation} \\
F(1 + (hM+m) \div (iK+(K-1)+1))
\end{cases} \]
\[ x[K-1] \]

Hence, we obtain the following program text

\[ \{ \bar{z} = \bar{c}_h(i) \land \bar{y} = \bar{f}_h(i) \land R \} \]
\[ m := 0 \\
; \text{do } m \neq M \rightarrow \\
x[0], x[x[m]] := \text{CASE}(x[0], x[x[m]], y_2), x[K-1] \\
; p := 1 \\
; \text{do } p \neq K \rightarrow \\
x[p] := \text{CASE}(x[p], x[p-1], y[p-1]) \\
; p := p + 1 \\
\text{od} \\
; m := m + 1 \\
\text{od} \\
; y_2 := y[K-1] \\
\{ \bar{z} = \bar{c}_{h+1}(i) \land \bar{y} = \bar{f}_{h+1}(i) \land R_{h+1}^i \} \]

The total number of variables needed for channels $\bar{c}_h$ and $\bar{f}_h$ is $M+2K+1$. Which is significantly better than the number of $4M+4K$ variables that the coarse-grained program needs when the construction of Appendix A would have been applied. \[\square\]
Chapter 6

Experimental Results

In this chapter, we discuss experimental results obtained from implementations on both VLSI and a processor network. For a processor network implementation, the execution speed is considered to be the most important performance figure. We distinguish two cases. In one case each process is allocated at its own processor and in the other case a number of processes is assigned to each processor. For VLSI implementations, chip area is an important characteristic. Area should be minimized as long as the execution speed meets a predefined speed requirement.

6.1 Processor Networks

In this section, the experimental results obtained from processor network implementations of both running examples are presented. The network on which the experiments have been carried out is a transputer network containing 51 transputers of type IMS T800-25. Each transputer has 1 Megabyte off-chip memory. Since both running examples have a linear topology, it suffices to configure the transputers into a ring topology. Implementations are written in the programming language Pascal, which was extended with some primitives for expressing communication along links and for defining processes [14].

In Section 2.6 and Chapter 3, we postulated a number of conjectures on the performance of processor network implementations. The validity of these conjectures is verified by the experiments that are discussed in this section. It turns out that, with respect to the quantitative results, the experiments are not completely in accordance with the theoretical results. This is mainly caused by the fact that one does not have much influence on the memory allocation of program variables. For a more detailed discussion, we refer to Section 6.1.3. For now, we discuss whether the results confirm our expectations in a qualitative sense or not.

First, results of implementations in which each process is allocated at its own processor are discussed. Next, we consider implementations in which each processor executes a number of processes. In this case, processor-links often share a number of channels. Therefore, multiplexing and de-multiplexing processes have to be added to the program. The complexity of these additional (de)multiplexing processes depends on the properties of the processes that communicate with each other. Without going into details, we mention that the amount of overhead introduced by (de)multiplexors
Table 6.1: Experimental results for a processor network implementation of the OCL problem. Time in seconds. Number of processors: 31. Problem size $N = 51 \times M$.

depends on the program at hand.

We recall that coarse-grained processes are composed of $M$ fine-grained processes and that each message sent along a channel contains $K$ (single) output values. Parameter $H$ is the quotient of $M$ and $K$. Parameter $N$ denotes the problem size and $L$ denotes the number of processes that are assigned to each processor.

6.1.1 A Single Process per Processor

We have implemented the OCL problem as an example of a pipe-line computation. For such a computation, the influence of choosing different values for parameter $K$ on the performance of a fixed problem (fixed $M$) is investigated. For the Palindrome Recognizer, as an example of a stack-like computation, it is determined whether splitting the computation into two parts of equal time complexity increases the efficiency of a program (cf. Optimization 2 in Section 2.6). This method for reducing idle time of a computation has only been implemented for the single process per processor case, since allocating more than one process at each processor is a more natural way of reducing idle time.

Pipe-line Computations

Experimental results of an implementation of the OCL problem are listed in Table 6.1. Four different problem sizes ($N$) are considered. For each problem, a number of values for parameter $K$ were chosen. The time for producing 30000 (single) output values has been measured. From the results, we infer that choosing $K$ equal to $M$ yields an optimal performance for $M=30$ and $M=12$. For $M=60$ and $M=20$, however, $K=M$ does not yield the best execution times. Since the deviation from the optimal execution times is within acceptable limits (5% and 3%, respectively), the results indicate that choosing $K$ equal to $M$ leads to efficient programs.

Stack-like Computations

As an example of a stack-like computation, the program for palindrome recognition has been implemented. Each process (cf. Example 2.20) is idle when its neighboring
6.1. Processor Networks

<table>
<thead>
<tr>
<th>$K$</th>
<th>no. of processes</th>
<th>'normal computation'</th>
<th>'split computation'</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1250</td>
<td>2.47</td>
<td>1.55</td>
</tr>
<tr>
<td>10</td>
<td>500</td>
<td>4.45</td>
<td>2.46</td>
</tr>
<tr>
<td>20</td>
<td>250</td>
<td>7.86</td>
<td>4.19</td>
</tr>
<tr>
<td>25</td>
<td>200</td>
<td>9.57</td>
<td>4.96</td>
</tr>
</tbody>
</table>

Table 6.2: Experimental results for a processor network implementation of the Palindrome Recognizer. Comparison between a 'normal' implementation and an implementation in which the computation is equally divided between communications. Time in seconds. $M = K$. Problem size $N = 10000.$

processes are performing computations. A more efficient computation can be obtained by splitting the computation into two parts such that both take approximately the same amount of time. Thereby, idle time can be reduced. Table 6.2 gives the results of a comparison between the original program and an implementation in which the computation is equally divided between communications. Since the number of processors is restricted to 51, each implementation consists of the first 51 processes only. On account of the regularity of the overall program, however, we conjecture that the measured results are as accurate as the results would have been for a processor network of a size that is large enough to accommodate all processes. For reasons of simplicity, we only considered the $[K, K]$-transformation.

The results confirm that the execution time can be decreased by equally dividing the computation between the communications. As expected, the larger parameter $K$ the larger the impact on the execution time. For $K=25$, the execution time has decreased by almost a factor of 2.

There is no need for splitting the computation if multiple processes are assigned to the each processor, since processes in the ready queue take over control when a process becomes idle. In the next section, we, therefore, consider results obtained from the original program for palindrome recognition, only.

6.1.2 Multiple Processes per Processor

Despite the overhead introduced by multiplexing and de-multiplexing processes, allocating more than one process at each processor usually leads to more efficient programs, since idle time of processors can be reduced.

In the analyses of Chapter 3 it was assumed that a processor network has the following properties

- a fast process-switch
- overhead of (de-)multiplexors can be neglected

Transputers do have a fast process-switch, since in their design, it was recognized that a fast process-switch is vital for parallel processing. However, in a transputer network,
the multiplexing of channels is not supported by dedicated hardware. Thus, it is the programmer’s responsibility to add multiplexing and de-multiplexing processes to the program. The overhead introduced by these processes cannot be neglected. As a result, we expect that the analyses of Chapter 3 are too optimistic.

For the OCL problem, we investigate the conjecture that the best performance is obtained by allocating only a few processes at each processor.

An analysis of the behavior of stack-like computations, like the Palindrome Recognizer, is much more difficult than the analysis of pipe-line computations. Problem size $N$ determines both the number of (single) output values to be computed and the number of fine-grained processes. Therefore, parameter $M$ also determines the number of coarse-grained process that are allocated to each processor: for problem size $N$, $M+L$ is a constant. From this observation, we infer that the larger $M$, the less communication overhead and, moreover, the less parallelism. Hence, there is a trade-off between communication overhead and available parallelism. Another important aspect of this type of computation is the fact that the time in which each process is active differs a lot among the processes of the computation, viz. the first process is active during the entire computation, whereas the last process’s activity is only small. In this respect, the behavior of stack-like computations is different from the behavior of pipe-line computations in which each process is, apart from a short start-up phase, active during the entire computation.

**Pipe-line Computations**

Results from an implementation of the OCL problem are listed in Table 6.3. Parameter $L$ denotes the number of processes that reside on a single processor. As in the single process per processor case, the time for producing 30000 output values has been measured. The results for $L=1$ are taken from Table 6.1. These experiments confirm the conjecture that assigning only a few processes to each processor gives the best performance. We also notice that the measured times for $L=3$ and for $L=4$ are only a fraction slower than the optimal execution time for $L=2$.

**Stack-like Computations**

Experimental results for an implementation of the Palindrome Recognizer, our running example, have been carried out for four different problem sizes. The results for $N=2400$, $N=6000$, $N=12000$, and $N=60000$ are listed in Table 6.4, Table 6.5, Table 6.6, and Table 6.7, respectively.

From the experiments for small problem sizes, we infer that in case of the optimal program, parameters $K$ and $M$ are equal ($N=2400$ yields $[K, M]=[8, 8]$ and $N=6000$ yields $[K, M]=[15, 15]$). For larger problem sizes, parameter $K$ and $M$ of the optimal problem are different from each other. In these cases, however, there are programs, for which $K=M$ holds, that have an execution time that differs from the optimal execution time by only a small factor (for $N=12000$, the execution time of $[K, M]=[15, 15]$ is only 1.6% larger than the execution time of $[K, M]=[24, 12]$, and for $N=60000$, the execution time of $[K, M]=[75, 15]$ and $[K, M]=[75, 75]$ differ by 4.7%). The results
6.1. Processor Networks

<table>
<thead>
<tr>
<th>$M$</th>
<th>$L$</th>
<th>$H(K = M/H)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>60</td>
<td>1</td>
<td>8.60</td>
</tr>
<tr>
<td>30</td>
<td>2</td>
<td>6.94</td>
</tr>
<tr>
<td>20</td>
<td>3</td>
<td>6.97</td>
</tr>
<tr>
<td>15</td>
<td>4</td>
<td>6.97</td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>7.23</td>
</tr>
<tr>
<td>10</td>
<td>6</td>
<td>7.71</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>10.44</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>12.53</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
<td>16.39</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>24.38</td>
</tr>
</tbody>
</table>

Table 6.3: Experimental results for a processor network implementation of the OCL problem. $L$: number of processes per processor. Time in seconds. Number of processors: 51. Constant problem size $N = 3060$.

suggest that for an optimal execution time the number of processes per processor should increase as the problem size grows.

A sequential program for palindrome recognition has also been implemented. This program was based on the parallel program. From the sequential execution times, speedup and efficiency of the parallel programs can be computed. For $N=2400$, the sequential execution time is 3.73 seconds, resulting in a speedup of 17.8 and an efficiency of 35%. The sequential execution time for the other instances is 23.22 seconds, 92.76 seconds, and (an estimated) 2320 seconds for $N=6000$, $N=12000$, and $N=60000$, respectively, resulting in speedups of 26.1, 30.7, and 35.45 and efficiencies of 51%, 60%, and 70%. These results confirm the well-known fact that larger problems yield a larger efficiency [11].

<table>
<thead>
<tr>
<th>$M$</th>
<th>$L$</th>
<th>$H(K = M/H)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td>1</td>
<td>0.37</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>0.22</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>0.21</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>0.23</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>0.32</td>
</tr>
</tbody>
</table>

Table 6.4: Experimental results for a processor network implementation of the Palindrome Recognizer. $L$: number of processes per processor. Time in seconds. Number of processors: 51. Problem size $N = 2400$. 
### Table 6.5: Experimental results for a processor network implementation of the Palindrome Recognizer. $L$: number of processes per processor. Time in seconds. Number of processors: 51. Problem size $N = 6000$.

<table>
<thead>
<tr>
<th>$M$</th>
<th>$L$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>10</th>
<th>15</th>
<th>60</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>1</td>
<td>2.17</td>
<td>2.06</td>
<td>2.03</td>
<td>2.02</td>
<td>2.01</td>
<td>2.01</td>
<td>2.03</td>
<td>2.44</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>2</td>
<td>1.16</td>
<td>1.12</td>
<td>1.12</td>
<td>1.15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>3</td>
<td>1.01</td>
<td>0.95</td>
<td></td>
<td>1.01</td>
<td>1.05</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>4</td>
<td>0.89</td>
<td></td>
<td>1.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.92</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>0.90</td>
<td>0.98</td>
<td>1.10</td>
<td>1.20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>6</td>
<td>0.94</td>
<td>1.07</td>
<td></td>
<td>1.58</td>
<td>2.43</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>1.25</td>
<td>1.66</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>1.49</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>15</td>
<td>1.88</td>
<td>2.89</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 6.6: Experimental results for a processor network implementation of the Palindrome Recognizer. $L$: number of processes per processor. Time in seconds. Number of processors: 51. Problem size $N = 12000$.

<table>
<thead>
<tr>
<th>$M$</th>
<th>$L$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>120</td>
<td>1</td>
<td>8.57</td>
<td>8.07</td>
<td>7.91</td>
<td>7.83</td>
<td>7.79</td>
<td>7.79</td>
<td>7.74</td>
<td>7.73</td>
<td>7.73</td>
<td>7.83</td>
</tr>
<tr>
<td>60</td>
<td>2</td>
<td>6.26</td>
<td>4.18</td>
<td>4.13</td>
<td>4.10</td>
<td>4.10</td>
<td>4.10</td>
<td>4.16</td>
<td>4.20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>3</td>
<td>4.38</td>
<td>3.63</td>
<td></td>
<td>3.42</td>
<td>3.37</td>
<td></td>
<td>3.47</td>
<td>3.56</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>4</td>
<td>3.64</td>
<td>3.21</td>
<td>3.11</td>
<td>3.22</td>
<td>3.29</td>
<td></td>
<td>3.49</td>
<td>3.58</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>5</td>
<td>3.38</td>
<td>3.02</td>
<td>3.06</td>
<td>3.18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>6</td>
<td>3.19</td>
<td>3.03</td>
<td></td>
<td>3.30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>8</td>
<td>3.07</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>10</td>
<td>3.40</td>
<td>3.55</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>3.58</td>
<td>4.01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>15</td>
<td>4.06</td>
<td>4.91</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
6.1. Processor Networks

<table>
<thead>
<tr>
<th>$M$</th>
<th>$L$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>600</td>
<td>1</td>
<td>215.93</td>
<td>203.31</td>
<td>116.50</td>
<td>102.61</td>
<td>109.89</td>
<td>195.40</td>
</tr>
<tr>
<td>300</td>
<td>2</td>
<td>124.98</td>
<td>129.62</td>
<td>116.50</td>
<td>102.61</td>
<td>109.89</td>
<td>82.39</td>
</tr>
<tr>
<td>200</td>
<td>3</td>
<td>107.41</td>
<td>103.43</td>
<td>82.39</td>
<td>85.56</td>
<td>85.56</td>
<td>85.56</td>
</tr>
<tr>
<td>150</td>
<td>4</td>
<td>100.64</td>
<td>84.76</td>
<td>92.24</td>
<td>84.76</td>
<td>92.24</td>
<td>92.24</td>
</tr>
<tr>
<td>120</td>
<td>5</td>
<td>94.40</td>
<td>84.40</td>
<td>65.62</td>
<td>69.37</td>
<td>74.62</td>
<td>74.62</td>
</tr>
<tr>
<td>100</td>
<td>6</td>
<td>76.09</td>
<td>72.80</td>
<td>65.83</td>
<td>68.33</td>
<td>68.33</td>
<td>68.33</td>
</tr>
<tr>
<td>75</td>
<td>8</td>
<td>68.73</td>
<td>67.07</td>
<td>65.83</td>
<td>65.45</td>
<td>65.45</td>
<td>65.45</td>
</tr>
<tr>
<td>60</td>
<td>10</td>
<td>71.35</td>
<td>68.21</td>
<td>69.38</td>
<td>69.38</td>
<td>69.38</td>
<td>69.38</td>
</tr>
<tr>
<td>50</td>
<td>12</td>
<td>74.50</td>
<td>72.32</td>
<td>72.32</td>
<td>72.32</td>
<td>72.32</td>
<td>72.32</td>
</tr>
</tbody>
</table>

Table 6.7: Experimental results for a processor network implementation of the Palindrome Recognizer. L: number of processes per processor. Time in seconds. Number of processors: 51. Problem size $N = 60000$.

6.1.3 Implementation Details

The transputer is a processor that is suited for application in processor networks on account of its fast process-switch and on-chip dedicated hardware for communication. In the experiments, we used a type of transputer that has 4 kBytes of on-chip random access memory. On-chip memory is much faster than off-chip memory. Thus, the performance of a program depends on the memory allocation of program variables. Variables that are often accessed should be allocated in on-chip memory (the OCCAM-compiler, for instance, always maps replicator variables onto on-chip memory). The existence of fast on-chip and slower off-chip memory has not been taken into account in the performance analyses that were presented in Section 2.6 and Chapter 3. By controlling memory allocations—in a naive way—we can, nevertheless, verify these theoretical results.

To be able to verify the sequence function of the $[M,K]$-transformations, we implemented a program such that the memory allocation for each $[M,K]$-transformation is the same. This is established as follows: each declaration of a variable of type array $[1..MI]$ (resp. $K$) is replaced by a declaration of type array $[1..UB]$ where $UB$ is a large constant and the program only accesses the first $M$ elements. This strategy has been applied to the OCL problem. Experimental results of both this implementation and the ‘normal’ implementation are listed in Table 6.8 (single process per processor). We notice a considerable distinction between the results of the ‘fixed’ and the ‘normal’ implementation. As expected, the results for the ‘normal’ implementation are the best. More importantly, the results of the ‘fixed’ implementation are in accordance with the sequence function of Chapter 3, i.e. from the execution times of the 13 experiments parameters $\alpha_0$, $\alpha_1$, and $\beta$ can be determined with an accuracy of 98%. (We note that implementations in OCCAM (instead of Pascal) are also in accordance with the corresponding sequence function [28].)
<table>
<thead>
<tr>
<th>$M$</th>
<th>$L$</th>
<th>'fixed'</th>
<th>'normal'</th>
</tr>
</thead>
<tbody>
<tr>
<td>90</td>
<td>90</td>
<td>7.13</td>
<td>6.36</td>
</tr>
<tr>
<td>45</td>
<td>90</td>
<td>7.93</td>
<td>6.36</td>
</tr>
<tr>
<td>30</td>
<td>90</td>
<td>8.42</td>
<td>6.37</td>
</tr>
<tr>
<td>18</td>
<td>90</td>
<td>9.71</td>
<td>6.07</td>
</tr>
<tr>
<td>60</td>
<td>60</td>
<td>5.30</td>
<td>4.30</td>
</tr>
<tr>
<td>30</td>
<td>60</td>
<td>6.27</td>
<td>4.10</td>
</tr>
<tr>
<td>20</td>
<td>60</td>
<td>7.23</td>
<td>4.10</td>
</tr>
<tr>
<td>15</td>
<td>60</td>
<td>8.19</td>
<td>4.10</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
<td>4.12</td>
<td>2.13</td>
</tr>
<tr>
<td>15</td>
<td>30</td>
<td>6.04</td>
<td>2.13</td>
</tr>
<tr>
<td>10</td>
<td>30</td>
<td>7.95</td>
<td>2.13</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>4.36</td>
<td>1.48</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>7.23</td>
<td>1.48</td>
</tr>
</tbody>
</table>

Table 6.3: Experimental results for a processor network implementation of the OCL problem. Comparison between the execution times of an implementation with a fixed memory allocation and the 'normal' implementation used before. Time (in seconds) for producing 15000 single output values. Number of processors: 51. Problem size $N = 51 * M$.

6.2 VLSI

In this section, the performance of VLSI implementations of parallel programs is discussed. The results are obtained from experiments that have been carried out with the DICY-system [29], a silicon compiler. Programs are written in the programming language Tangram. The compilation of Tangram programs into VLSI is syntax-directed. Thereby, the programmer has control over certain properties (e.g., area and speed) of the design. For example, the number of variables that are used in a program is a good estimate for the area of the resulting VLSI circuit. Hence, the introduction of an additional variable usually leads to an increase in chip area. The timing results that we present are obtained by simulating the behavior of the chip, not by testing a concrete VLSI chip. Experiences with the DICY-system indicate, however, that the simulation results are quite accurate.

6.2.1 Simulation Results

Tangram programs have been written for the OCL-problem. The program has been obtained by applying an $[M, 1]$-transformation as given in Section 4.1. Two different implementations are considered. In the first solution the array that each process of the program contains has been implemented by a cyclic array; in the second solution the array has been implemented by a collection of (simple) variables. If $M+1$ is a power of 2, the array pointer in the 'cyclic array' implementation can be implemented very
efficiently. We therefore compare both implementations only for such parameters $M$.

The experimental results for the ‘cyclic array’ implementation and the ‘variables’ implementation are listed in Table 6.9. In the column ‘area’ the chip area of a single process of the program is listed. Due to the structure of the processes — to be more precise: there is no multiplexing of input or output ports — the overall chip area of a program can be obtained by multiplying the chip area of a single process by the number of processes of the program. In the column ‘time’ the cycle time of a single step in the repetition is listed.

### 6.2.2 Area

The number of variables in a program is a good estimate for the chip area of the resulting VLSI design. Both implementations for the OCL problem have a number of variables that is linear in parameter $M$. This yields the following equation for chip area as a function of parameter $M$.

$$A = \gamma_0 M + \gamma_1$$

For the ‘cyclic array’ implementation this equation gives $\gamma_0 = 0.13$ and $\gamma_1 = 0.33$ with an accuracy of 98% for $M = 3, 7, 15, 31$. The ‘variables’ solution gives $\gamma_0 = 0.13$ and $\gamma_1 = 0.53$ with an accuracy of 98%.

These results confirm that an $[M, 1]$-transformation is an effective way of decreasing chip area. Moreover, the ‘variables’ implementation is slightly cheaper than the ‘cyclic array’ implementation. A small penalty is paid for the introduction of an array (only parameter $\gamma_1$ differs for both solutions).

### 6.2.3 Cycle Time

In Table 6.9, the cycle times of the two implementations of the OCL problem are listed. For the ‘cyclic array’ implementation one step in the repetition consists of the
following computation:

\[ b?vb, c?V[w], d?vd \]
\[ ; w := w + 1 \]
\[ ; ( \text{if } m \neq w \rightarrow vb := vb + [V[m] = vd] \mid m = w \rightarrow \text{skip fi} \]
\[ ; m := m + 1 \]
\[ ; \text{ff} \]
\[ ; bbvbc, cc!V[w], ddvd \]

And for the 'variables' implementation one step in the repetition consists of

\[ b?vb, c?vc, d?vd \]
\[ ; v_M := vc \]
\[ ; vc := v_0 \]
\[ ; v_0 := v_1 \]
\[ ; \ldots \]
\[ ; v_{M-1} := v_M ; vb := vb + [v_M = vd] \]
\[ ; v_M := v_{M-2} ; vb := vb + [v_M = vd] \]
\[ ; v_M := v_{M-3} ; vb := vb + [v_M = vd] \]
\[ ; \ldots \]
\[ ; v_M := v_0 ; vb := vb + [v_M = vd] \]
\[ ; bbvbc, cc!vc, ddvd \]

Consider the 'cyclic array' implementation first. From the program text, we infer that the cycle time is linear in parameter \( M \):

\[ t = \alpha + M \beta \]

The simulation results of Table 6.9 yield \( \alpha = 0.330 \) and \( \beta = 0.233 \) with an accuracy of 95%.

For the timing results of the 'variables' solution we have to take into account the multiplexer that the silicon compiler introduces for variable \( v_M \). The number of stages in the multiplexer grows linearly with parameter \( M \), resulting in a cycle time that is quadratic in parameter \( M \):

\[ t = \alpha_0 + M \beta_0 + M^2 \beta_1 \]

The simulation results yield \( \alpha = 8.76 \times 10^{-3}, \beta_0 = 0.152, \) and \( \beta_1 = 5.8 \times 10^{-3} \) with an accuracy of 99%.

In Figure 6.1 chip area and cycle time are depicted for both implementations. Chip area is relative to the area that a single fine-grained process would occupy. For example, in the 'cyclic array' implementation, for \( M = 3 \) the relative chip area of a single fine-grained process is 0.40 (\( = 1.21/3 \)). The trade-off between area and cycle time is controlled by the choice for parameter \( M \) (see also [13]).
Figure 6.1: cycle time vs. relative area (• = 'cyclic array' implementation; o = 'variables' implementation)
Chapter 7

Concluding Remarks

In this thesis, we addressed the topic of designing parallel programs. The design of such a program consists of two steps:

1. Design a fine-grained parallel program from its specification.

2. Apply an $[M, K]$-transformation to this fine-grained program in order to obtain a coarse-grained parallel program.

The second design phase yields a parallel program of parameterized granularity. When designing a parallel program one often does not know which grain size will give an optimal performance [2, 9, 18]. Therefore, it is advantageous to introduce parameters that control the grain size. Parameterized granularity also offers the possibility of implementing a program on a variety of machines by choosing appropriate values for the parameters depending on the characteristics of a particular machine.

Although Chapter 5 shows that a coarse-grained program can be obtained from a fine-grained program in a systematic way, it is recommended to design programs of parameterized grain size by starting from parameterized i/o-relations that reflect the i/o-relations of the original fine-grained program. In this way, experience gained in the construction of fine-grained programs can be utilized and, moreover, problem specific information can be incorporated in the design.

At first, the $[M, K]$-transformation was introduced for designing programs that run on processor networks in particular. However, this technique turned out to be an effective way of reducing chip area of VLSI implementations of parallel programs, too. Throughout this thesis we therefore discussed both processor network implementations and VLSI implementations.

The design of a good fine-grained program is the stepping-stone to a good coarse-grained program. Therefore, we paid much attention to a solid design technique for fine-grained programs. It is, in particular, important that from the problem’s specification a good set of equations, that the values communicated along the input and the output channels satisfy, is derived. The derivation of the set of equations often makes an appeal to the programmer’s creativity. Methods that aim at automated synthesis of VLSI processor arrays (which can be considered to be fine-grained programs) usually start from such a set of equations or from some initial set on which a number of transformations are applied to obtain a ‘feasible’ set of equations. Our
method for designing fine-grained programs was inspired by the method presented in
[8, 16, 25, 30, 32], but differs with respect to the following:

- In our method, i/o-relations are expressed in terms of the specification's global
  streams. The other method makes no distinction between global streams and
  local streams, which gives rise to i/o-relations in which the output streams of a
  process are directly related to the input streams of the same process.

- The notational conventions of the other method give rise to linear topologies
  and tree topologies, only. Furthermore, global input and output is handled by
  a single process. The notational conventions in our method do not restrict the
  class of network topologies that can be described. It also does not restrict the
  location of the global streams.

It is our experience that many parallel programs can be derived in a more natural way
in our method than in the other method.

In Practice

From a practical point of view it is important to choose appropriate values for pa-
rameters \( M \) and \( K \) once we have obtained an \([M, K]\)-transformation. As usual, we
distinguish VLSI implementations and processor network implementations.

VLSI  When implementing a parallel program as a VLSI circuit it is important that
chip area is minimized provided that the speed requirements are met. A good
estimate for chip area is the number of program variables (cf. Section 6.2). Choos-
ing \( K = 1 \) yields a minimal number of variables. When a program has to be imple-
mented as a VLSI circuit, one should, therefore, apply an \([M, 1]\)-transformation.

Processor Networks  Section 6.1 shows that the execution speed of a processor net-
work implementation of an \([M, K]\)-transformation depends on the processor's
characteristics. In the ideal case, a processor should have a fast process-switch
dedicated hardware for communication (including routers and (de-)multiplexors),
and fast memory access.

Theoretical results imply that for processor network implementations one should
apply a \([K, K]\)-transformation (this is confirmed by the experimental results of
Section 6.1.3). However, due to specific architectural details of the processor, it
is sometimes better to apply an \([M, K]\)-transformation where \( M \neq K \).

The allocation of multiple processes at each processor, instead of one process,
leads to efficient programs. As a rule of thumb for obtaining the best results,
one should map only a few (2–4) processes per processor for pipe-line programs.
For stack-like programs, the number of processes per processor should increase
when the problem size increases. In this kind of computation there is a trade-
off between the available parallelism (more processes per processor increases
available parallelism) and communication overhead (less processes per processor
reduces the communication overhead).
More Than One Dimension

For simplicity’s sake, we restricted ourselves to programs of a one-dimensional structure. Of course, more-dimensional fine-grained programs can be transformed into coarse-grained programs as easily as one-dimensional fine-grained programs. We confine ourselves to giving an example of a 2-dimensional problem.

Consider a collection of processes in a 2-dimensional grid. Process \((k_0, k_1), 0 \leq k_0\) and \(0 \leq k_1\), has input channels \(a_{k_0,k_1+1}\) and \(b_{k_0,k_1+1}\) and output channels \(a_{k_0,k_1}\) and \(b_{k_0,k_1}\); see Figure 7.1. Assume that process \((k_0, k_1)\) exhibits communication behavior

\[
(a_{k_0+1,k_1}, b_{k_0,k_1+1}; a_{k_0,k_1}, b_{k_0,k_1})^-
\]

Let \(a_{k_0,k_1}(i) = F_{k_0,k_1}(i)\) and \(b_{k_0,k_1}(i) = E_{k_0,k_1}(i)\) be the i/o-relations of channels \(a_{k_0,k_1}\) and \(b_{k_0,k_1}\), respectively. We now compose \(M^2\) processes into one large process, such that process \((k_0, k_1)\) is the composition of fine-grained processes \((h_0,M+p_0, h_1,M+p_1)\) where \(0 \leq p_0, p_1 < M\). As an example, output channels \(a_{k_0,M+h_0,M+p_0}\) of the fine-grained processes are multiplexed into output channel \(a_{k_0,k_1}\) (cf. Figure 7.2).

Each communication along channel \(a_{k_0,k_1}\) deals with \(KM\) communications of the original fine-grained program. The i/o-relation for channel \(a_{k_0,k_1}\) is

\[
\tilde{a}_{k_0,k_1}(i)[p,q] = E_{h_0,h_1,M+q}(iK+p)
\]

since \(a_{k_0,k_1}(i)[p,q] = a_{k_0,M+h_0,M+q}(iK+p)\), for \(0 \leq p < K\) and \(0 \leq q < M\).

According to the communication behavior of the fine-grained processes, we have

\[
a_{k_0+1,k_1} < a_{k_0,k_1}\]

For the coarse-grained process we then obtain

\[
a_{k_0,M+h_0,M+p} < \cdots < a_{k_0,M+h_1,M+q}(iK+p)
\]

Hence, \(a_{k_0+1,k_1} < \tilde{a}_{k_0,k_1}\). Therefore, the communication behavior of the parameterized coarse-grained program is

\[
(\tilde{a}_{k_0+1,k_1}, \tilde{b}_{k_0+1,k_1}; \tilde{a}_{k_0,k_1}, \tilde{b}_{k_0,k_1})^-
\]

This example shows that applying an \([M,K]\)-transformation to a 2-dimensional fine-grained program yields a coarse-grained program of the same structure. A coarse-grained process is composed of \(M^2\) fine-grained processes and messages consist of packets of \(KM\) single values. In [26] a \([K,K]\)-transformation has been applied to a fine-grained program for dynamic programming.
Figure 7.2: coarse-grained process in a 2-dimensional grid
Bibliography


Appendix A

General Scheme for Stack-like Computations

In this appendix we demonstrate the applicability of an \([M, K]\)-transformation to obtain a coarse-grained program for any stack-like computations. From Section 5.2 we recapitulate the set of equations that channels \(a_k\) and \(b_k\), and local variables \(u_k\) and \(w_k\) of fine-grained process \(k\) satisfy:

\[
\begin{align*}
    b_k(i) &= F_k(a_k(i), b_{k+1}(i-L), u_k(i), w_k(i)) \\
    a_{k+1}(i) &= G_k(a_k(i+L-1), u_k(i+L-1)) \\
    v_k(i+1) &= S_k(a_k(i), u_k(i)) \\
    w_k(i+1) &= T_k(a_k(i), b_{k+1}(i-L), u_k(i), w_k(i))
\end{align*}
\]

Consider coarse-grained process \(h\) that is a composition of fine-grained processes \(hM_1, \ldots, hM+M-1\). As shown in Section 4.2, the communication behavior of this process is

\[
\bar{a}_h ; (\bar{b}_h ; \bar{a}_h)^{LH-1} ; (\bar{b}_h, \bar{a}_{h+1} ; \bar{a}_h, \bar{b}_{h+1})^r
\]

where \(H= M/K\). As usual, values communicated along channels \(\bar{a}_h\) and \(\bar{b}_h\) satisfy

\[
\begin{align*}
    \bar{a}_h(i)[p] &= a_{hM}(iK+p) \\
    \bar{b}_h(i)[p] &= b_{hM}(iK+p)
\end{align*}
\]

for \(i \geq 0\) and \(0 \leq p < K\).
We construct a coarse-grained program that has the following structure

\[
[x : \text{array } [0..K) \text{ of } T_0; \tilde{x} : \text{array } [0..M) \text{ of } T_0; \\
\bar{u} : \text{array } [0..K) \text{ of } T_1; \bar{v} : \text{array } [0..M) \text{ of } T_2; \\
\bar{w} : \text{array } [0..M) \text{ of } T_3;
\]

initialize \(R_0^i\)

\[
\begin{align*}
&\{x = \tilde{a}_h(i) \land R\} \\
&S_0 \\
&\{\bar{u} = \tilde{b}_h(i) \land R_{i+1}^i\} \\
&\bar{b}_h \mid \bar{u} \\
&\bar{a}_h ? \tilde{x} \land \{LH-1\}
\end{align*}
\]

\[
\begin{align*}
&\{x = \tilde{a}_h(i) \land \bar{u} = b_{h+1}(i-LH) \land R\} \\
&S_1 \\
&\{x = a_{h+1}(i-LH+1) \land \tilde{u} = \tilde{b}_h(i) \land R_{i+1}^i\} \\
&\bar{b}_h \mid \bar{u}, \bar{a}_{h+1} ? \tilde{x} \\
&\bar{a}_h ? \tilde{x}, \bar{b}_{h+1} ? \bar{u}
\end{align*}
\]

where \(R\) is defined as

\[
\begin{align*}
R_0 : & \quad \tilde{x}[p] = a_{hM+p+1}(iK-(p+1)L) \\
R_1 : & \quad \bar{v}[p] = v_{hM+p}(iK-pL) \\
R_2 : & \quad \bar{w}[p] = w_{hM+p}(iK-pL)
\end{align*}
\]

for \(0 \leq p < M\). Statement \(S_1\) is an extension of statement \(S_0\), since in \(S_1\) postcondition \(\tilde{x} = \tilde{a}_{h+1}(i-LH+1)\) has to be established also. We, therefore, consider the derivation of statement \(S_1\), which can be decomposed into three parts: \(S', S'', S'''\), such that \(S_1 = S', S'', S'''\).

\[
\begin{align*}
\{x = \tilde{a}_h(i) \land \bar{u} = \tilde{b}_h(i-LH) \land R\} \\
S' \\
\{x = \tilde{a}_h(i) \land I\} \\
S'' \\
\{x = \tilde{a}_h(i) \land J\} \\
S''' \\
\{x = \tilde{a}_{h+1}(i-LH+1) \land \tilde{u} = \tilde{b}_h(i) \land R_{i+1}^i\}
\end{align*}
\]
A.1 Statement $S'$

where $I$ is defined as

$$
I_0: \quad z[p] = \begin{cases} 
    a_{M+p+1}(iK-(p+1)L) & \text{for } 0 \leq p < M-K \\
    v_{M+p}(iK-pL) & \text{for } 0 \leq p < M-K \\
    u_{M+p}(iK+p-M+K-pL) & \text{for } M-K \leq p < M \\
    w_{M+p}(iK-pL) & \text{for } 0 \leq p < M-K \\
    w_{M+p}(iK+p-M+K-pL) & \text{for } M-K \leq p < M \\
    b_{h+1}(i-LH) & \text{for } 0 \leq p < K
\end{cases}
$$

and $J$ is defined as

$$
J_0: \quad z[p] = \begin{cases} 
    a_{M+K+p+1}((i+1)K-(K+p+1)L) & \text{for } 0 \leq p < M-K \\
    v_{M+p}(iK+p-pL) & \text{for } 0 \leq p < K \\
    u_{M+p}((i+1)K-pL) & \text{for } K \leq p < M \\
    w_{M+p}(iK+p-pL) & \text{for } 0 \leq p < K \\
    w_{M+p}((i+1)K-pL) & \text{for } K \leq p < M \\
    b_{h+1}(i-K-(p+1)L) & \text{for } 0 \leq p < K
\end{cases}
$$

A.1 Statement $S'$

Statement $S'$ of the program should satisfy

$$\{x=a_h(i) \land \bar{u} \equiv \bar{b}_{h+1}(i-LH) \land R\} \implies \{x=a_h(i) \land I\}$$

To establish the postcondition from the precondition, we introduce variable $m$ and accompanying invariant $P$:

$$
P_0: \quad z[p] = \begin{cases} 
    a_{M+p+1}(iK-(p+1)L) & \text{for } 0 \leq p < M-m \\
    v_{M+p}(iK-pL) & \text{for } 0 \leq p < M-m \\
    u_{M+p}(iK+p-M+m-pL) & \text{for } M-m \leq p < M \\
    w_{M+p}(iK-pL) & \text{for } 0 \leq p < M-m \\
    w_{M+p}(iK+p-M+m-pL) & \text{for } M-m \leq p < M \\
    b_{h+1}(i-LH) & \text{for } 0 \leq p < m
\end{cases}
$$

Since $P_0^m$ is equivalent to $\bar{u} = \bar{b}_{h+1}(i-LH) \land R$, invariant $P$ is initially satisfied by $m:=0$. The postcondition is implied by $P_K^m$. Establishing $P_{m+1}^m$ from $P$ requires a repetition with invariant $Q$ and the introduction of variable $n$ (let $r$ and $s$ denote $M-m+n$ and
\( M - m - 1 \), respectively:

\[
\begin{align*}
Q_0: & \quad z[s] = a_{K+M+r} \cdot (iK+M+rL) \\
Q_1: & \quad u[p] = \left\{ \begin{array}{ll}
u_{K+M+r} & (iK+p-s-pL) \\
v_{M+r} & (iK+p-s-1-pL) \end{array} \right. \quad \text{for } M-m \leq p < r \\
Q_2: & \quad w[p] = \left\{ \begin{array}{ll}
w_{K+M+r} & (iK+p-s-pL) \\
w_{M+r} & (iK+p-s-1-pL) \end{array} \right. \quad \text{for } r \leq p < M \\
Q_3: & \quad u[p] = \left\{ \begin{array}{ll}
u_{K+M+r} & (iK+p-s-pL) \\
u_{M+r} & (iK+p-s-1-pL) \end{array} \right. \quad \text{for } r \leq p \leq p < m \\
\end{align*}
\]

Invariant \( Q \) is established by \( m := 0 \), using \( P \), and \( P_{m+1}^m \) is implied by \( Q_{m}^n \). Taking into account that \( r = M - m + n \) yields the following derivation for establishing \( (Q_0)^n_{m+1} \) from \( Q_0 \)

\[
\begin{align*}
a_{K+M+r} & = \text{defining equation for } a_k \\
G_{K+M+r} & = (a_{K+M+r}(iK+n-rL), v_{K+M+r}(iK+n-rL)) \\
& \quad \{ Q_0 \text{ and } Q_1 \} \\
G_{K+M+r} & = (z[s], v[r]) \\
\end{align*}
\]

Similar derivations can be given for \( Q_1, Q_2, \) and \( Q_3 \), resulting in the following program text for \( S' \) (let \( f_+ \) be a shorthand for \( F_{K+M+r} \cdots \)):

\[
\begin{align*}
\{ \bar{z}=\bar{a}_k(i) \land \bar{u}=\bar{b}_{k+1}(i-L.H) \land R \} \\
; m, s := 0, M-1 \quad \{ P \} \\
; m \neq K \rightarrow \\
; n, r := 0, s+1 \quad \{ Q \} \\
; m \neq m \rightarrow \\
; \overline{z}[s], \overline{u}[n], \overline{v}[r], \overline{w}[r] := \\
\{ \bar{z}_1, \bar{z}_2, \bar{z}_3 \} \\
; n, r := n+1, r+1 \quad \{ Q_{m+1} \} \\
\end{align*}
\]

\[
\begin{align*}
\{ \bar{z}=\bar{a}_k(i) \land I \}
\end{align*}
\]

### A.2 Statement \( S'' \)

Statement \( S'' \) of the program should satisfy

\[
\{ \bar{z}=\bar{a}_k(i) \land I \} \tilde{S}'' \{ \bar{z}=\bar{a}_k(i) \land J \}
\]
A.2. Statement $S''$

Invariant $P$ and variable $m$ are introduced in order to establish the postcondition from the precondition.

$$P_0: \quad z[p] = \begin{cases} a_{hM+p+1}(iK-(p+1)L) & \text{for } 0 \leq p < m \\ a_{hM+K+p+1}(i+1)K-(K+p+1)L) & \text{for } m \leq p < M-K \\ v_{hM+p}(iK-pL) & \text{for } 0 \leq p < m \\ v_{hM+K+p}(i+1)K-pL) & \text{for } m+K \leq p < M \\ \end{cases}$$

$$P_1: \quad v[p] = \begin{cases} v_{hM+p}(iK+p-m-pL) & \text{for } m \leq p < m+K \\ v_{hM+p}(i+1)K-pL) & \text{for } m+K \leq p < M \\ \end{cases}$$

$$P_2: \quad w[p] = \begin{cases} w_{hM+p}(iK-pL) & \text{for } 0 \leq p < m \\ w_{hM+K+p}(i+1)K-pL) & \text{for } m+K \leq p < M \\ \end{cases}$$

$$P_3: \quad u[p] = b_{hM+m+p+1}(iK+p-(m+p+1)L) \quad \text{for } 0 \leq p < K$$

Since $I$ implies $P_{M-K}$, invariant $P$ is initially satisfied by $m := M-K$. Postcondition $J$ is implied by $P^m_K$. Establishing $P$ from $P_{m+1}$ requires a repetition with invariant $Q$ and the introduction of variable $n$ (let $r$ denote $m+n+1$):

$$Q_0: \quad z[m] = a_{hM+r}(iK+n-rL)$$

$$Q_1: \quad v[p] = \begin{cases} v_{hM+p}(iK+p-m-pL) & \text{for } m \leq p \leq r-1 \\ v_{hM+p}(iK+p-(m+1)-pL) & \text{for } r-1 \leq p \leq m+K \\ \end{cases}$$

$$Q_2: \quad w[p] = \begin{cases} w_{hM+p}(iK+p-m-pL) & \text{for } m \leq p \leq r-1 \\ w_{hM+p}(iK+p-(m+1)-pL) & \text{for } r-1 \leq p \leq m+K \\ \end{cases}$$

$$Q_3: \quad u[p] = \begin{cases} b_{hM+m+p+1}(iK+p-(m+p+1)L) & \text{for } 0 \leq p < n \\ b_{hM+m+1+M+1}(iK+p-(m+1+p+1)L) & \text{for } n \leq p < K \\ \end{cases}$$

Invariant $Q$ is established by $n := 0$, using $P^m_{m+1}$. Moreover, $Q^K_m$ implies $P$. Taking into account that $r = m+n+1$ yields the following derivation for establishing $(Q_2)_{n+1}$ from $Q_2$:

$$w_{hM+r}(iK+r-m-rL)$$

$$= \{ \text{defining equation for } u_k \}$$

$$T_{M+r}(a_{hM+r}(iK+n-rL), b_{hM+r+1}(iK+n-(r+1)L), v_{hM+r}(iK+n-rL), w_{hM+r}(iK+n-rL))$$

$$= \{ Q \}$$

$$T_{M+r}(z[m], u[n], v[r], w[r])$$

Similar derivations can be given for $Q_0$, $Q_1$, and $Q_3$. This results in the following
program text (again, let \( f \) be a shorthand for \( F_{kM+r} \)):

\[
\{ \bar{z} = \bar{a}_k(i) \land J \} \\
M := M - K \quad \{ P \} \\
\text{do } m \neq 0 \rightarrow \\
m, n, r := m - 1, 0, m + n + 1 \quad \{ P_{m+1}^n \land Q \} \\
\text{do } n \neq K \rightarrow \\
z[m], u[n], v[r], w[r] := \\
\text{} \quad g_s(z[m], v[r]), f_s(z[m], u[n], v[r]), u[n], v[r]), \\
\text{} \quad s_s(z[m], v[r]), t_s(z[m], u[n], v[r]), w[r]) \quad \{ Q_{n+1}^r \} \\
\text{od} \quad \{ P \} \\
\text{od} \quad \{ P_0^n \} \\
\{ \bar{z} = \bar{a}_k(i) \land J \}
\]

### A.3 Statement \( S''' \)

Statement \( S''' \) of the program should satisfy

\[
\{ \bar{z} = \bar{a}_k(i) \land J \} S'' \{ \bar{z} = \bar{a}_{k+1}(i - LH + 1) \land \bar{u} = \bar{b}_k(i) \land R_{i+1}^i \}
\]

Statement \( S''' \) is split into three parts:

\[
\{ \bar{z} = \bar{a}_k(i) \land J \} \\
S''_0 \quad \{ J' \} \\
S''_1 \quad \{ \bar{u} = \bar{b}_k(i) \land R_{i+1}^i \} \\
S''_2 \quad \{ \bar{z} = \bar{a}_{k+1}(i - LH + 1) \land \bar{u} = \bar{b}_k(i) \land R_{i+1}^i \}
\]

where \( J' \) equals \( J_0 \land J_1 \land J_2 \land J_3 \) with

\[
J_0: \quad z[p] = \begin{cases} 
\bar{a}_{AM}((i+1)K - (p + 1)) & \text{for } 0 \leq p < K \\
\bar{a}_{AM+p+1}((i + 1)K - (p + 1)L) & \text{for } K \leq p < M 
\end{cases}
\]

Notice that \( z[p] = z[K - p - 1] \) for \( 0 \leq p < K \). If array \( \bar{z} \) is implemented as a cyclic array, the implementation of \( S''_0 \) is simple. We therefore focus our attention on statements \( S''_1 \) and \( S''_2 \). Consider statement \( S''_1 \) first.

To establish postcondition \( \bar{u} = \bar{b}_k(i) \land R_{i+1}^i \) from precondition \( J' \) invariant \( P \) and variable
A.3. Statement $S^m$

$m$ are introduced

\[
P_0: \quad \begin{cases} 
    a_{hM}((i+1)K-(p+1)) & \text{for } 0 \leq p < K-m \\
    a_{hM+p+1}((i+1)K-(p+1)L) & \text{for } K-m \leq p < M \\
    v_{hM+p}(iK+m+p-pL) & \text{for } 0 \leq p < K-m \\
    v_{hM+p}((i+1)K-pL) & \text{for } K-m \leq p < M \\
    w_{hM+p}(iK+m+p-pL) & \text{for } 0 \leq p < K-m \\
    w_{hM+p}((i+1)K-pL) & \text{for } K-m \leq p < M \\
    b_{hM}(iK+p) & \text{for } 0 \leq p < m \\
    b_{hM+p-m+1}(iK+p-(p-m+1)L) & \text{for } m \leq p < K 
\end{cases}
\]

Invariant $P$ is initially satisfied by $m:=0$. Furthermore, $P^P$ implies postcondition $\bar{a}=-\bar{b}(i) \wedge R_{x+1}$. Establishing $P^m_{m+1}$ from $P$ requires a repetition with invariant $Q$ and the introduction of variable $n$ (let $r$ and $s$ denote $n-m$ and $K-m-1$, respectively):

\[
Q_0: \quad z[s] = a_{hM+r}(iK+n-rL) \\
Q_1: \quad \begin{cases} 
    v_{hM+p}(iK+m+p+1-pL) & \text{for } 0 \leq p < r \\
    v_{hM+p}(iK+m+p-pL) & \text{for } r \leq p < K-m \\
    w_{hM+p}(iK+m+p+1-pL) & \text{for } 0 \leq p < r \\
    w_{hM+p}(iK+m+p-pL) & \text{for } r \leq p < K-m \\
    b_{hM+p-m}(iK+p-(p-m)L) & \text{for } m \leq p < n \\
    b_{hM+p-m+1}(iK+p-(p-m+1)L) & \text{for } n \leq p < K 
\end{cases}
\]

Invariant $Q$ is established by $n:=m$, using $P$. Moreover, $Q^P_K$ implies $P^m_{m+1}$. Taking into account that $r=n-m$ yields the following derivation for establishing $(Q_3)_{m+1}^n$ from $Q_3$:

\[
\begin{align*}
    &b_{hM+r}(iK+n-rL) \\
    &= \{ \text{defining equation for } b_k \} \\
    &= F_{hM+r}(a_{hM+r}(iK+n-rL), b_{hM+r+1}(iK+n-(r+1)L) \\
    &\quad, v_{hM+r}(iK+n-rL), w_{hM+r}(iK+n-rL)) \\
    &= \{ Q \} \\
    &= F_{hM+r}(z[s], u[n], v[r], w[r])
\end{align*}
\]

Similar derivations can be given for $Q_0$, $Q_1$, and $Q_2$, resulting in the following program
(let \( f \) be a shorthand for \( F_{n,M+r,...} \)):

\[
\begin{align*}
\{ J \} \\
& m, s := 0, K - 1 & \{ P \} \\
& \text{do } m \neq K \rightarrow \\
& \quad n, r := m, 0 & \{ Q \} \\
& \text{do } n \neq K \rightarrow \\
& \quad \begin{align*}
& \quad z[s], u[n], v[r], w[r] := \\
& \quad \quad g_\sigma(z[s], v[r]), f_r(z[s], u[n], v[r], w[r]), \\
& \quad \quad s_r(z[s], u[n], v[r], w[r]) \quad \{ Q_{n+1}^m \} \\
& \quad ; n, r := n + 1, r + 1 \\
& \text{od} & \{ P_{m+1}^n \} \\
& \text{od} & \{ P_{m+1}^n \} \\
& m, s := m + 1, s - 1 \\
& \text{od} & \{ P_{m+1}^n \} \\
& \{ \bar{u} = \bar{a}_0(i) \land R_{i+1}^n \}
\end{align*}
\]

Finally, we consider statement \( S_{n+1}^m \) which computes \( x = a_{n+1}(i-LH+1) \) and leaves \( R_{i+1} \) invariant. For this computation we introduce invariant \( P \) and an auxiliary array \( \bar{y} \) of size \( K \).

\[
\begin{align*}
P_0 : & \quad x[p] = \begin{cases} 
a_{(h+1)m + ((i-LH+1)K+p)} & \text{for } 0 \leq p < m \\
a_{(h+1)m - ((i+1)K-(M-p)L)} & \text{for } m \leq p < K 
\end{cases} \\
P_1 : & \quad y[p] = \begin{cases} 
v_{(h+1)m - K+p-(i+1)K-(M-p)L)} & \text{for } K-m \leq p < K 
\end{cases}
\end{align*}
\]

Provided that \( x[p] = a_{(h+1)m + ((i+1)K-(M-p)L)} \), for all \( 0 \leq p < K \), is established, invariant \( P \) is initialized by \( m := 0 \). According to \( R_{i+1}^n \), we obtain \( x[p] = x[M-p-1] \). The postcondition is implied by \( P_{m+1}^n \) for establishing \( P_{m+1}^n \) we design a repetition with invariant \( Q \) and introduce variable \( n \) (let \( r \) denote \( K-m-n \)):

\[
\begin{align*}
Q_0 : & \quad x[m] = a_{(h+1)m+K+r} ((iK+m+r)-(M-K+r)L) \quad \text{for } 0 \leq n < m \\
Q_1 : & \quad y[p] = \begin{cases} 
v_{(h+1)m+K+r} ((iK+p+m+1)-(M-K+p)L) & \text{for } r-n-1 \leq p < r \\
v_{(h+1)m+K+r} ((iK+p+m)-(M-K+p)L) & \text{for } r \leq p < K 
\end{cases}
\end{align*}
\]

Using \( R_{i+1}^n \land P, Q_0^m \) holds provided that \( y[K-m-1] = y[M-m-1] \). Postcondition \( P_{m+1}^n \) is implied by \( Q_{m+1}^n \). Taking into account that \( r = K-m+n \) we derive for establishing \( (Q_0)_{m+1}^n \) from \( Q_0 \)

\[
\begin{align*}
& a_{(h+1)m+K+r+1} ((iK+m+r+1)-(M-K+r+1)L) \\
& \quad \{ \text{defining equation for } a_k \} \\
& \quad \quad G_{(h+1)m+K+r} (a_{(h+1)m+K+r+1} ((iK+m+r+1)-(M-K+r+1)L)) \\
& \quad \quad \quad \quad ; v_{(h+1)m+K+r} ((iK+m+r)-(M-K+r)L)) \\
& \quad \{ Q \} \\
& \quad G_{(h+1)m+K+r} (x[m], y[r])
\end{align*}
\]
A similar derivation can be given for $Q_1$. We obtain the following program (let $g_{M-K+r}$ be a shorthand for $G_{bM+M-K+r}$)

\[
\{u=b_h(i) \land R_{i+1}\} \\
\text{\hspace{1cm}}\text{\texttt{\texttt{n := 0; do n K \rightarrow z[n] := z[M-n-1]; n := n + 1 od}}}
; m := 0 \{} \{P\} \\
; \text{\texttt{do m K \rightarrow}} \\
\text{\hspace{1cm}}\text{\texttt{y[K-m-1] := v[M-m-1]}} \\
\text{\hspace{1cm}}\text{\texttt{n, r := 0, K-m \{} \{Q\} \\
; \text{\texttt{do n K \rightarrow}}} \\
\text{\hspace{2cm}}\text{\texttt{x[m], y[r] := g_{M-K+r}(x[m], y[r]), s_{M-K+r}(x[m], y[r]) \{Q_{n+1}\}}} \\
\text{\hspace{2cm}}\text{\texttt{n, r := n + 1, r + 1}} \\
\text{\hspace{1cm}}\text{\texttt{od \{} \{P_{m+1}\} \\
; m := m + 1 \\
\text{\hspace{1cm}}\text{\texttt{od \{} \{P_{R}\} \\
\text{\hspace{1cm}}\text{\texttt{\{u=b_h(i) \land \bar{R}_{i+1}\}}}}
\]

A.4 An Optimization

The set of defining equations for a general stack-like program contains two local variables $v_h(i)$ and $w_h(i)$. The value of local variable $v_h(i)$ is only determined by the values communicated along input channel $a_k$ and is used to determine values for output channel $a_{k+1}$. This has been the sole purpose for introducing $v_h(i)$. However, there is often a very simple relation between values communicated along $a_k$ and $a_{k+1}$, like

\[a_{k+1}(i) = G_h(a_k(i+L-1))\]

or like

\[a_{k+1}(i) = a_k(i+L-1)\]

for all $i \geq 0$. In both cases statement $S''_2$ can be simplified to a large extent, since array $y$ does not have to be introduced. In the latter case, which very often occurs, statement $S''_2$ simply boils down to

\[
\text{\texttt{n := 0; do n K \rightarrow x[n] := x[M-n-1]; n := n + 1 od}}
\]
Index

chip area, 30, 69
coarse-grained, 3, 39, 72
code size, 30
command, 8
communication behavior, 8, 11, 23, 42, 46
communication overhead, 35, 38
communication processor, 36
correct by design, 2
cyclic array, 45
data processor, 36
data size, 30, 45, 50, 51
data-independent, 3, 11
deadlock, 9, 26
efficiency, 31, 35, 38, 39, 41, 62
fine-grained, 2, 35, 72
input/output relation, 7, 13, 43, 46
latency, 11, 15, 31
\([M,K]\)-transformation, 39, 42, 52, 72
memory allocation, 67
minimal buffering, 9, 25, 47
more-dimensional, 74
multiplexing, 61, 63
negatively indexed element, 12, 23
parallelism
  algorithmic, 2
  geometric, 2
partial order, 8, 25, 46
passive clustering, 36
processor farming, 2, 13
program notation, 6
projection, 9
response time, 10, 31
sequence function, 10, 30, 39
set of equations, 7, 17, 53, 56, 72
speedup, 31
stream, 11
unspecified value, 21
window computation, 12
Samenvatting

Het onderwerp van dit proefschrift is het ontwerpen van parallelle programma’s. In het bijzonder wordt de implementatie van parallelle programma’s op processorsnetwerken beschouwd, waarbij iedere processor een eigen deel van het programma (een proces) uitvoert. Tijdens de berekening wisselen processoren onderling informatie uit door het versturen van boodschappen over kanalen.

In hoofdstuk 2 wordt een methode behandeld om uitgaande van een specificatie op een systematische wijze een parallel programma af te leiden dat aan die specificatie voldoet. De volgens deze methode ontworpen programma’s hebben typisch een fijnkorrelige structuur, hetgeen resulteert in processen die weinig tijd besteden aan berekeningen ten opzichte van de tijd die aan communicatie met buurprocessen besteed wordt. De communicatieoverhead van zulke processen is groot. Dit heeft als resultaat dat dergelijke programma’s doorgaans niet efficiënt geïmplementeerd kunnen worden op een processorsnetwerk.

De communicatieoverhead kan op een systematische wijze verkleind worden door programma’s te ontwerpen waarvan de korrelgrootte geparametriserd is. Door in een dergelijk programma een geschikte keuze te doen voor de parameters kan een efficiënte implementatie worden verkregen. Een dergelijke geschikte keuze zal afhankelijk zijn van de eigenschappen van het processorsnetwerk waarop het programma wordt geïmplementeerd. Geïmplementeerde programma’s lenen er zich ook voor om het benodigde chipopervlak van VLSI-implementaties te beïnvloeden.

De programma’s die beschouwd worden zijn ‘stream-oriented’ en de processen hebben een regelmatig dataafhankelijk communicatiegedrag. In dit proefschrift wordt uitgegaan van twee typen parallelle programma’s. Het ene type programma gedraagt zich als een pipe-line en het andere type programma gedraagt zich als een stack.

De gepropageerde methode om parallelle programma’s met een geparametriseerde korrelgrootte af te leiden valt uiteen in twee stappen:

1. Ontwerp, uitgaande van de specificatie, een fijnkorrelig programma.

2. Transformeer het fijnkorrelige programma tot een programma met een geparametriseerde korrelgrootte.

Bij de tweede stap kan gebruik worden gemaakt van de kennis die is opgedaan bij het ontwerpen van het fijnkorrelige programma in de eerste stap.

In hoofdstuk 6 wordt de toepasbaarheid van de theorie geverifieerd door experimentele resultaten. Daarbij worden zowel implementaties op een processorsnetwerk als VLSI-implementaties beschouwd.
Curriculum vitae

1. Voor de toepasbaarheid van grote processornetten is een goede verhouding tussen de capaciteiten van de (data)processor en de communicatiehardware van belang. De trend dat in nieuwe generaties processoren de verbetering van de capaciteit van de (data)processor een factor hoger is dan de verbetering van de capaciteit van de communicatiehardware leidt tot een afname van het aantal processoren in nieuwe generaties processornetten.

2. Indien voor een branch and bound algoritme, zoals bijvoorbeeld voor het Handelsreisigersprobleem, een efficient algoritme bestaat om de ondergrens van de kostenfunctie van een bepaald deelprobleem te berekenen, dan zal het distribueren van deelproblemen over beschikbare processoren niet leiden tot een efficient parallel algoritme. In een dergelijk geval dient het berekenen van de ondergrens van de kostenfunctie geparallelliseerd te worden.


3. De in dit proefschrift gebruikte notatie voor het ontwerpen van parallele programma's is geïnspireerd door de zogenaamde 'p-punt' notatie. Laatstgenoemde notatie is echter beperkend en daardoor minder geschikt in die zin dat de notatie aansluit op het ontwerpen van een lineaire of een boomtopologie waarbij de (globale) in- en outputkanalen zich aan dezelfde kant van het netwerk bevinden. Ook het feit dat bij de 'p-punt' notatie de specificatie van de i/o-relatie van outputkanalen gerelateerd wordt aan de waarden die een proces over zijn inputkanalen ontvangt, leidt vaak tot minder manipulatievrijheid voor de ontwerper.


4. Bestaande technieken om fijnkorrelige programma's — genaamd VLSI processor arrays — te construeren hebben vaak een recursieve betrekking als uitgangspunt. Het ontwerp van een goede betrekking wordt in deze methoden onderbelicht, hoewel dit juist aan de creativiteit van de ontwerper appelleert.
5. Het non-deterministisch gedrag van parallelle programma's maakt dat voor het ontwerp van zulke programma's formele methoden onontbeerlijk zijn. Alleen dit feit al staat brede toepassing van parallelle computers buiten onderzoeksstellingen in de weg.

6. De in dit proefschrift beschreven \([M,1]\)-transformatie is een effectief middel om het benodigde chipoppervlak voor VLSI-implementatie te verkleinen.

Lit.: - Hoofdstuk 6 van dit proefschrift

7. Zij \( N = \sum_{k=0}^{K} n_k 2^k \), met \( n_k \in \{0,1\} \) en \( n_K = 1 \). Definieer \( M_k := N \text{ div } 2^k \). Indien proces \( k, 0 \leq k < K \), de volgende programmatekst heeft:

\[(a_k?; b_k!0)^{\ast k}; (a_k?; b_k!0, a_{k+1}!; a_k?, b_{k+1}?y; b_k!1)^{M_k+1}\]

en proces \( K \) de volgende programmatekst heeft:

\[(a_K?; b_K!1)\]

dan geldt \( b_0(i) = 0 \) voor \( 0 \leq i < N-1 \) en \( b_0(N-1) = 1 \).

Een sequence functie voor dit programma is

\[
\sigma(a_k,i) = i \ast 2^{k+1} + f_k \\
\sigma(b_k,i) = i \ast 2^{k+1} + g_k
\]

met \( f_0 = 0, g_0 = 1 \), \( f_{k+1} = n_k 2^{k+1} + g_k \) en \( g_{k+1} = n_k 2^{k+1} + f_k \).

Herhaald gedrag van bovenstaand programma levert een modulo-\( N \) counter.


8. Het helder en zo begrijpelijk mogelijk uittegen van onderzoeksresultaten leidt er, ten onrechte, vaak toe dat de kwaliteit van het onderzoek wordt onderschat.

9. Het is schadelijk voor het prestige van een sport als er ook een gelijknamig campingspel bestaat, dat naast het hebben van dezelfde naam weinig overeenkomsten met die sport heeft.