TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Techniques for designing efficient parallel programs

Citation for published version (APA):
Struik, P. (1991). *Techniques for designing efficient parallel programs*. (Computing science notes; Vol. 9132). Technische Universiteit Eindhoven.

Document status and date:
Published: 01/01/1991

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 23. Aug. 2022

Eindhoven University of Technology

Department of Mathematics and Computing Science

Techniques for Designing Efficient Parallel Programs

by

Pieter Struik

91/32

Computing Science Note 91/32
Eindhoven, December 1991

COMPUTING SCIENCE NOTES

This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology.
Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review.
Copies of these notes are available from the author.

Copies can be ordered from:
Mrs. F. van Neerven
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
ISSN 0926-4515

All rights reserved
editors: prof.dr.M.Rem
prof.dr.K.M.van Hee.

Techniques for Designing Efficient Parallel Programs

Pieter Struik
Department of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, the Netherlands

Abstract

In this paper we present techniques for designing parallel programs. These techniques are calculational, i.e. starting from a formal specification of the problem we design a program by transforming the specification in a number of steps. The programs that we obtain are correct by design. We demonstrate two techniques by means of an example. First, we illustrate the derivation of a fine grained program. Since the communication overhead of such a program is too large, it can not be implemented on a processor network (e.g. transputer network) efficiently. We, subsequently, demonstrate two techniques for designing programs of a parameterized grain size. For obtaining efficient execution, such programs offer the possibility of tuning to the characteristics of the machine on which it is executed. Finally, we give a complexity analysis of the techniques presented and compare the results from the analysis to experimental results obtained from a transputer implementation.
keywords: design techniques, parallel programming, grain size

1 Introduction

In this paper we present techniques for deriving parallel programs of parameterized grain size. We consider a parallel program to be a collection of processes that interact with each other by exchanging messages. For efficient execution of parallel programs it is important that the partitioning of programs into processes is done properly. In particular, the grain size of processes is important. Processes that perform a lot of computations between successive communications with other processes are said to have a large grain size.

When designing a parallel program it is a difficult to determine a grain size that yields a good performance when the program is executed [1, 2]. It is therefore advantageous to

The work in this paper has been partly sponsored by Philips Research Laboratories Eindhoven, the Netherlands

postpone this design decision by designing a parallel program that has a parameterized grain size. Such a program offers the possibility of tuning it to the characteristics of the machine on which it is executed.

Much research has been done on extracting parallelism from sequential programs [3]. In our technique, however, we follow the opposite direction. Starting from a fine grained parallel program we construct programs of parameterized grain size.

This paper is organized as follows. In Section 2 we briefly discuss a technique for deriving fine grained parallel programs. This technique is illustrated by means of a so-called window computation, viz. the Occurrence Count Last problem (OCL). We conclude Section 2 by giving a short complexity analysis of the program. In Section 3 the basic idea of this paper is presented. Based on the program derived in Section 2, we construct parallel programs of parameterized grain size for the OCL problem. We demonstrate two techniques. In Section 4, we present some experimental results obtained from a transputer implementation of the problem. Experiments have been carried out on a 51-transputer network[1]. The results will be related to the complexity analysis worked out in Section 3. Finally, Section 5 gives some concluding remarks.

2 Design of a fine grained parallel program

In this section, we present the derivation of a fine grained program for the OCL problem. Starting from a specification, we construct a linear array of processes that communicate with each other by exchanging messages over channels. We derive a set of equations that define the values communicated along output channels in terms of received values along input channels. Since inputs on which an output depends should be received before producing that output, the set of defining equations gives rise to a partial order on communications along channels. Given this partial order we construct a consistent communication behavior that specifies in which order communications along the channels of a process take place. Given the communication behavior and the set of equations, the program text of a process can easily be written. We conclude this section with a short complexity analysis of the constructed program. Since the topic of this paper is not on the design of fine grained programs, we only briefly discuss the design method. For more examples of the design method we refer to [4, 5, 6]. The derivation of a fine grained program for a particular problem constitutes a basis that can be fruitfully exploited when designing a program of parameterized grain size.

2.1 Specification of the OCL problem

For a fixed N ($N \geq 1$) and an input stream A of integers, the OCL problem is the computation of output stream B satisfying

$$B(i) = (\# j : i-N < j \leq i : A(j) = A(i))$$

[1]acquired through a grant from the European Community, Parallel Computing Action of ESPRIT, PCA No. 4038

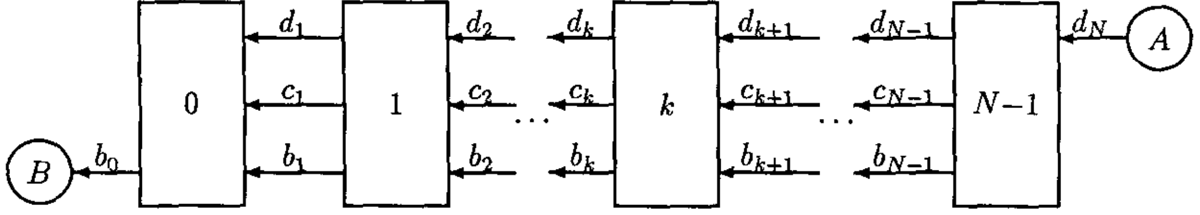

Figure 1: linear arrangement of processes for the OCL problem

for $i \geq 0$, where $(\# j : R : B.j)$ denotes the number of $j \in R$ satisfying boolean expression $B.j$. Elements of streams are indexed from 0.

The OCL problem is called a window computation since $B(i)$ is determined by the N elements of window $A(i-N..i]$. Parameter N is called the window length. In this computation, from each window of length N of the input stream A it is computed how many times the last element of that window occurs in it. For $i<N$, the window contains negatively indexed elements of input stream A. In the sequel, we assume $A(-j)=0$ for $j>0$. Each successive element of stream B is obtained by computing a function on the previous window that is shifted over one position. As a result, the program can produce one element of the output stream for each element of the input stream it receives.

2.2 A fine grained program for the OCL problem

We construct a fine grained program that consists of a linear network of N processes, numbered from 0 (see Figure 1). Each process k $(0 \leq k < N)$ has an output channel b_k specified as follows

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

The specification of channel b_k is a generalization of the specification of output stream B. Notice that process 0 produces output stream B, since $b_0(i) = B(i)$.

For process $N-1$ we have $b_{N-1}(i)=1$ and for the other processes we derive

$$\begin{array}{ll} & b_k(i) \\ = & \quad \{\text{specification } b_k\} \\ & (\# j : i-N+k < j \leq i : A(j) = A(i)) \\ = & \quad \{\text{split off term } j=i-N+k+1\} \\ & [A(i-N+k+1) = A(i)] + (\# j : i-N+k+1 < j \leq i : A(j) = A(i)) \\ = & \quad \{\text{specification } b_{k+1}\} \\ & [A(i-N+k+1) = A(i)] + b_{k+1}(i) \end{array}$$

where [true] = 1 and [false] = 0.

From this derivation we infer that in order to compute $b_k(i)$ process k needs to have at its disposal two elements of the global input stream A, viz. $A(i-N+k+1)$ and $A(i)$. The

indices of both elements are equal for process $k{=}N{-}1$. Hence, process $N{-}1$ should access input stream A. We, therefore, decide that $A(i{-}N{+}k{+}1)$ and $A(i)$ are communicated to process k by process $k{+}1$ and introduce two additional output channels for process k

$$\begin{aligned} c_k(i) &= A(i-N+k) \\ d_k(i) &= A(i) \end{aligned}$$

For the first output along channel c_k we have $c_k(0){=}A(-N{+}k)$. Since negatively indexed element of A equal 0, we have $c_k(0){=}0$. Furthermore, we have $c_k(i+1){=}c_{k+1}(i)$ and $d_k(i) = d_{k+1}(i)$. Summarizing, we have the following equations for the output channels of process k ($k \neq N{-}1$)

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

A communication behavior of process k that is consistent with this set of equations and, moreover, introduces minimal buffering – i.e. a minimal number of variables per process – is

$$(b_{k+1}, c_{k+1}, d_{k+1}; b_k, c_k, d_k)^*$$

In this communication behavior, a semi-colon ';' denotes sequential composition, the Kleene star '*' denotes repetition, and a comma ',' between two communications denotes that we do not assign any particular order upon the execution (both actions may even be executed in parallel). Since neighbor processes access shared channels in the same order, deadlock is avoided.

Programs are written in a CSP-like notation, where $c?x$ ($c!x$) denotes the receipt (sending) of variable x along channel c. Translation of such a program into OCCAM is straightforward. The program text of process k reads

$$\begin{array}{l} \quad vcc := 0 \\ ;\ (\ \ b_{k+1}?vb, c_{k+1}?vc, d_{k+1}?vd \\ \quad ;\ vb, vc, vcc := vb + [vc = vd], vcc, vc \\ \quad ;\ b_k!vb, c_k!vc, d_k!vd \\ \quad)^* \end{array}$$

In this program we have four variables of type integer (vb, vc, vcc, and vd).

Although in an implementation on a processor network we are not that much interested in the number of variables a program uses, we nevertheless mention this number, since our programs can also be implemented as a VLSI circuit where chip area (heavily depending on the number of variables used) is one of the main design restrictions [7].

2.3 A short complexity analysis

We conclude this section with a short complexity analysis of the fine grained parallel program. We assume that each process is allocated to a separate processor. A so-called sequence function σ is used to analyse the time complexity of the program. $\sigma(e,i)$ denotes the time on which the i-th communication along channel e can be scheduled. Computations, denoted by τ, are also taken into account. On account of symmetry between channels b_k, c_k, and d_k, the communication behavior of process k (including computations) can be simplified to $(d_{k+1};\tau;d_k)^*$. As a complexity measure for a (concurrent) communication statement and for the computation, τ, we take α and β time units, respectively. We obtain the following sequence function

$$\sigma(d_k,i) = \alpha + (N-k)(\alpha+\beta) + i(2\alpha+\beta)$$

This sequence function is correct on account of $\sigma(d_k,i)-\sigma(d_{k+1},i)=\alpha+\beta$ and $\sigma(d_{k+1},i+1)-\sigma(d_k,i)=$
We are primarily interested in channel b_0. The time needed for the production of L outputs along channel b_0 is denoted by $t(L)$. Hence, $t(L) = \sigma(d_0, L-1)$ giving

$$t(L) = \alpha(N+2L-1) + \beta(N+L-1)$$

A sequential program takes approximately $s(L) = LN\beta$ time units. For $L \gg N$, our program has a speedup of $\frac{s(L)}{t(L)} = \frac{LN\beta}{2L\alpha+L\beta} = N(\frac{\beta}{2\alpha+\beta})$.

The efficiency of the program is defined by the quotient of the speedup and the number of processors used. In our example, N processors are used. We, therefore, obtain an efficiency of $\frac{\beta}{2\alpha+\beta} = \frac{1}{(2\alpha/\beta)+1}$.

Notice that the communication overhead $\frac{2\alpha}{\beta}$ determines both speedup and efficiency. The larger the communication overhead the lower the speedup of the parallel program. For $\frac{2\alpha}{\beta} \approx 0$, i.e. the communication time can be neglected in comparison with the computation time, we have an optimal efficiency of 1.

3 Design of coarse grained parallel programs

The complexity analysis of the previous section shows that the efficiency of a parallel program is determined by the communication overhead $\frac{2\alpha}{\beta}$. For the OCL problem a transputer implementation would, by counting the number of cycles a computation and communication take [8], typically give $\frac{2\alpha}{\beta} \approx 7$, yielding only $\frac{1}{8}$-th of the optimal efficiency. Although a transputer has a relatively efficient means of communication, the communication overhead of the fine grained program is too large for efficient execution (other processor networks suffer from even larger communication overheads). We, therefore, need a technique to reduce the communication overhead in order to design efficient parallel programs for processor networks. A VLSI implementation would typically give a communication overhead of $\frac{2\alpha}{\beta} \approx \frac{1}{2}$ [7].

In the remainder of this paper we discuss two techniques for reducing the communication overhead. By reducing the communication overhead we obtain parallel programs of a

coarser grain. One technique for enlarging the grain size of a parallel program is to compose larger processes from a number of processes, say M, of the fine grained program. This technique is called an $[M,1]$-transformation. The computation time within such a process increases by a factor M, giving a communication overhead of $\frac{2\alpha}{M\beta}$. A second technique for enlarging the grain size is a combination of composing larger processes and composing larger messages, i.e. messages that consist of a number of values that are communicated as a single packet. Assuming that a communication takes a communication setup time and a number of time units depending on the amount of data to be transferred, we are able to save $(K-1)$ times a communication setup time by transferring a single packet of K values instead of K single values. In the sequel, we will always compose K processes of the fine grained program when introducing packet size K, thereby expecting a communication overhead of at most $\frac{2\alpha}{K\beta}$. We refer to this technique as a $[K,K]$-transformation.

3.1 The $[M,1]$-transformation

In an $[M,1]$-transformation the computation time within a process is increased by composing a process from M processes of the fine grained program. The fine grained program for the OCL problem consists of N processes. As a result, the program obtained by applying an $[M,1]$-transformation consists of $\frac{N}{M}$ processes (assume that M is a divisor of N). With each process k $(0 \le k < \frac{N}{M})$ of the transformed program we associate an output channel B_k that is specified as

$$B_k(i) = (\# j : i-N+kM < j \le i : A(j) = A(i))$$

Notice that process 0 produces output stream B, since $B(i) = B_0(i)$. For process k $(k \ne \frac{N}{M}-1)$ we derive

$$\begin{array}{ll} & B_k(i) \\ = & \quad \{\text{specification } B_k\} \\ & (\# j : i-N+kM < j \le i : A(j) = A(i)) \\ = & \quad \{\text{split off } j\text{: } i-N+kM < j \le i-N+(k+1)M\} \\ & (\# j : i-N+kM < j \le i-N+(k+1)M : A(j) = A(i)) \\ & \ + (\# j : i-N+(k+1)M < j \le i : A(j) = A(i)) \\ = & \quad \{\text{rewrite range of quantification; specification } B_{k+1}\} \\ & (\# j : 0 < j \le M : A(kM+i-N+j) = A(i)) + B_{k+1}(i) \end{array}$$

From this relation we infer that $A(i)$ and $A(kM+i-N..(k+1)M+i-N]$ are needed for the computation of $B_k(i)$. We decide that $A((k+1)M+i-N)$ and $A(i)$ are communicated to process k by process $k+1$ along channels C_{k+1} and D_{k+1}. This gives rise to the following two specifications

$$\begin{array}{lll} C_k(i) & = & A(i-N+kM) \\ D_k(i) & = & A(i) \end{array}$$

Moreover, we introduce array V of dimension M local to process k satisfying

$$V(h) = A(i-N+kM+h)$$

for $0 \leq h < M$. With these definitions we come to the following structure of the program of process k in which only statement list S has to be determined

$$
\begin{array}{l}
m := 0;\ \mathbf{do}\ m \neq M \rightarrow V(m) := 0; m := m+1\ \mathbf{od} \\
; i := 0; \\
;\ (\ B_{k+1}?vb, C_{k+1}?vc, D_{k+1}?vd \\
\qquad \{V(h) = A(i-N+kM+h)\ \wedge\ vb = B_{k+1}(i)\ \wedge\ vc = C_{k+1}(i)\ \wedge\ vd = D_{k+1}(i)\} \\
\quad ;\ S \\
\qquad \{V(h) = A(i+1-N+kM+h)\ \wedge\ vb = B_k(i)\ \wedge\ vc = C_k(i)\ \wedge\ vd = D_k(i)\} \\
\quad ;\ B_k!vb, C_k!vc, D_k!vd \\
\quad ;\ i := i+1 \\
\quad)^*
\end{array}
$$

For statement list S we find

$$
\begin{array}{l}
vc, V(0) := V(0), vc \\
; m := 0;\ \mathbf{do}\ m \neq M-1 \rightarrow V(m), V(m+1) := V(m+1), V(m); m := m+1\ \mathbf{od} \\
; vb := vb + (\# j : 0 \leq j < M : V(m) = vd)
\end{array}
$$

Array V can more efficiently be implemented as a cyclic array. Therefore, the execution time of S is determined by the calculation of $B_k(i)$, i.e. assignment of variable vb.

In the above program we have, apart from some auxiliary variables, $M+3$ variables, viz. array V and variables vb, vc, and vd.

As expected, an $[M,1]$-transformation where $M=1$ yields a program that resembles the fine grained solution of the OCL problem.

3.2 The $[K,K]$-transformation

In a $[K,K]$-transformation not only the computation time is increased by composing larger processes out of processes of the fine grained solution, but also the communication time is decreased by composing larger messages. Communication time of a messages containing K values is modeled as $\alpha_0+K\alpha_1$, where $\alpha_0+\alpha_1$ is the time for communicating a single value: $\alpha_0+\alpha_1=\alpha$. Since K fine grained processes are composed, the number of processes in a $[K,K]$-transformation is $\frac{N}{K}$. Along output channel $\bar{b}_h$ of process h arrays of values are communicated. The p-th index of the array communicated in the i-th communication along channel $\bar{b}_h$ is denoted by $\bar{b}_h(i)[p]$ and specified as $(0 \leq h < \frac{N}{K})$

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

for $i \geq 0$ and $0 \leq p < K$. Notice that $\bar{b}_0(i)[p] = B(iK+p)$. In the derivation of a program for the $[K,K]$-transformation two additional output channels, $\bar{c}_h$ and $\bar{d}_h$, for process h have to

be introduced (cf. derivations of the fine grained program and the $[M,1]$-transformation). The specification of both channels is

$$\begin{aligned}\bar{c}_h(i)[p] &= A((i+h)K-N+p)\\ \bar{d}_h(i)[p] &= A(iK+p)\end{aligned}$$

Given these specifications we derive for $\bar{b}_h(i)[p]$ $(0\le h<\frac{N}{K}-1)$

$$\begin{aligned}
&\bar{b}_h(i)[p]\\
=&\quad \{\text{specification } \bar{b}_h\}\\
&(\#j : iK+p-N+hK < j \le iK+p : A(j) = A(iK+p))\\
=&\quad \{\text{split range of quantification; specification } \bar{b}_{h+1}\}\\
&(\#j : iK+p-N+hK < j \le iK+p-N+(h+1)K : A(j) = A(iK+p)) + \bar{b}_{h+1}(i)[p]\\
=&\quad \{\text{calculus}\}\\
&(\#j : 0 < j \le K \wedge p+j < K : A((i-1+h+1)K-N+(p+j)) = A(iK+p))\\
&+(\#j : 0 < j \le K \wedge K \le p+j < 2K : A((i+h+1)K-N+(p+j-K)) = A(iK+p))\\
&+ \bar{b}_{h+1}(i)[p]\\
=&\quad \{\text{specification } \bar{c}_{h+1} \text{ and } \bar{d}_{h+1}\}\\
&(\#j : 0 < j \le K \wedge p+j < K : \bar{c}_{h+1}(i-1)[p+j] = \bar{d}_{h+1}(i)[p])\\
&+ (\#j : 0 < j \le K \wedge p+j \ge K : \bar{c}_{h+1}(i)[p+j-K] = \bar{d}_{h+1}(i)[p])\\
&+ \bar{b}_{h+1}(i)[p]
\end{aligned}$$

The program text of process h for the $[K,K]$-transformation now boils down to

```
  p := 0; do p ≠ K → VCC(p) := 0; p := p + 1 od
; ( b̄_{h+1}?VB, c̄_{h+1}?VC, d̄_{h+1}?VD
  ; p := 0
  ; do p ≠ K
       →  VB[p] := VB[p] + (#j : 0 < j ≤ K ∧ p+j < K : VCC[p+j] = VD[p])
                         + (#j : 0 < j ≤ K ∧ p+j ≥ K : VC[p+j−K] = VD[p])
          ;p := p + 1
    od
  : VC, VCC := VCC, VC
  ; b̄_h!VB, c̄_h!VC, d̄_h!VD
  )*
```

where variables VB, VC, VCC, and VD are arrays of dimension K.

3.3 A complexity analysis

The programs of both the $[M,1]$-transformation and the $[K,K]$-transformation have the same structure as the fine grained program of Section 2.3. For the communication time and

computation time of the $[M,1]$-transformation we take α and $M\beta$ time units, respectively. Considering that the number of processes equals $\frac{N}{M}$, we have

$$\sigma(D_k,i) = \alpha + (\frac{N}{M}-k)(\alpha+M\beta) + i(2\alpha+M\beta)$$

The time for producing L outputs along channel B_0, denoted by $t_M(L)$, then is

$$t_M(L) = \sigma(D_0, L-1) = \alpha(\frac{N}{M}+2L-1) + \beta(N+LM-M)$$

resulting in a speedup of $\frac{LN\beta}{2L\alpha+LM\beta} = \frac{N}{M}(\frac{M\beta}{2\alpha+M\beta})$ and an efficiency of $(\frac{1}{(2\alpha/M\beta)+1})$, for $L \gg N$. The communication overhead for an $[M,1]$-transformation is $\frac{2\alpha}{M\beta}$, like we expected it to be.

For the communication time and computation time of the $[K,K]$-transformation we take $\alpha_0+K\alpha_1$ and $K^2\beta$ time units, respectively. Here, the number of processors equals $\frac{N}{K}$. We have

$$\sigma(\bar{d}_h,i) = (\alpha_0+K\alpha_1) + (\frac{N}{K}-h)((\alpha_0+K\alpha_1)+K^2\beta) + i(2(\alpha_0+K\alpha_1)+K^2\beta)$$

If the time needed for the production of L outputs along channel $\bar{b}_0$ is denoted by $t_K(L)$, we have

$$t_K(L) = \sigma(\bar{d}_0, \frac{L}{K}-1) = (\alpha_0+K\alpha_1)(\frac{N}{K}+2\frac{L}{K}-1) + \beta(NK+LK-K^2)$$

since $\frac{L}{K}$ messages contain L values. As a result, we obtain a speedup of $\frac{LN\beta}{2(L/K)(\alpha_0+K\alpha_1)+LK\beta} = \frac{N}{K}(\frac{K^2\beta}{2(\alpha_0+K\alpha_1)+K^2\beta})$ and an efficiency of $(\frac{1}{(2(\alpha_0+K\alpha_1)/K^2\beta)+1})$. As expected, the $[K,K]$-transformation yields a communication overhead of $\frac{2(\alpha_0+K\alpha_1)}{K^2\beta}$ $(\leq \frac{2K\alpha}{K^2\beta} = \frac{2\alpha}{K\beta})$.

Next, we consider the number of variables needed in the programs. Auxiliary variables are not considered. In the fine grained program, 4 variables have been declared for each process, resulting in a total number of $4N$ variables. A process of the $[M,1]$-transformation requires $M+3$ variables. This results in $\frac{N}{M}(M+3) = N+\frac{3N}{M}$ variables, since a program consists of $\frac{N}{M}$ processes. For the $[K,K]$-transformation, we obtain a total of $4N$ variables. In VLSI, one of the main design restrictions, chip area, depends mostly on the number of variables needed in the program and an $[M,1]$-transformation offers the possibility to reduce that number.

4 Experimental results

In this section we present the experimental results obtained from an OCCAM implementation of the OCL problem on a 51 T800-transputer network. Experimental results will be related to the complexity analysis of the previous section.

In the implementation each processor executes at most one process. This choice resembles the assumptions made in the complexity analysis and, thereby, makes verification of theoretical results by experiments possible. Allocating multiple processes at a single

processor requires another theoretical complexity analysis that is beyond the scope of this paper.

In the experiments, we do not measure the time for producing L output values, but for consuming L input values (i.e. we consider throughput). The reason for that is not fundamental, it only made the implementation simpler. When verifying the theoretical results we should consider the sequence function of the input channel, instead of the output channel.

4.1 Experimental Results for the $[M, 1]$-transformation

We carried out experiments for two problem instances, viz. one problem of problem size N=50 and one of problem size N=500. We measured the time for consuming L=100, 000 input values. The results are summarized is the following table (et = elapsed time in ms; pr = number of processes involved in the computation)

$N = 50$			$N = 500$		
M	et	pr	M	et	pr
1	2781	50	10	5301	50
2	3060	25	20	8267	25
5	3900	10	25	9707	20
10	5300	5	50	16899	10
25	9700	2	100	31299	5
50	16100	1	250	74499	2
			500	149742	1

Experimental results for $[M, 1]$-transformation

Notice the scalability of the algorithm by comparing the results that have the same parameter M. Notice, also, that the computation time increases, resulting is a smaller speedup. The efficiency, however, increases. As usual, there is a trade-off between speedup and efficiency.

For consumption of L input values, the theoretical complexity analysis gives

$$\sigma(D_{N/M}, L-1) = (2L-1)\alpha + M(L-1)\beta$$

Given this formula and the experimental results, we obtain 13 equations from which α and β can be deduced. We find

$$\begin{aligned} \alpha &= 12.33\,\mu s \\ \beta &= 2.88\,\mu s \end{aligned}$$

with an accuracy of more than 99%. As a result, the ratio $\frac{2\alpha}{\beta}$ equals 8.6 which comes close to the estimated communication overhead of $\frac{2\alpha}{\beta} \approx 7$, considering the fact that a

receiving process may not always want to accept a communication immediately. The sequential execution time, $LN\beta$, is 14.4 sec. and 144 sec. for problem size N=50 and N=500, respectively. As an example of the trade-off between speedup and efficiency, for problem size N=500 we have: M=10 gives a speedup of 27.1 and an efficiency of 54%, and M=100 gives a speedup of 4.6 and an efficiency of 92%.

4.2 Experimental results for the $[K,K]$-transformation

For the $[K, K]$-transformation the same experiments have been carried out as for the $[M, 1]$-transformation. The results are summarized in the following table ($L = 100,000$)

$N = 50$			$N = 500$		
K	et	pr	K	et	pr
1	2883	50	10	4017	50
2	2559	25	20	6457	25
5	2938	10	25	7667	20
10	4012	5	50	13739	10
25	7663	2	100	25912	5
50	13136	1	250	62593	2
			500	129550	1

Experimental results for $[K, K]$-transformation

For the consumption of L input values, the theoretical complexity analysis gives

$$\sigma(\bar{d}_{N/K}, \frac{L}{K}-1) = (2\frac{L}{K}-1)(\alpha_0+K\alpha_1) + K^2(\frac{L}{K}-1)\beta$$

We have 13 equations for parameters α_0, α_1, and β and obtain

$$\begin{aligned}\alpha_0 &= 5.77\,\mu s\\ \alpha_1 &= 7.42\,\mu s\\ \beta &= 2.45\,\mu s\end{aligned}$$

with an accuracy of more than 99%. Given $t_K(L)$, the time for producing L outputs, we infer that for large L the execution time is minimized for $K=\sqrt{\frac{2\alpha_0}{\beta}}$. In this case, we find K=2.2 . The experimental results indeed show that the execution time for K=2 is less than the execution time for K=1.

5 Concluding Remarks

In this paper we have presented two techniques for designing parallel programs of a parameterized grain size. Such a program offers the attractive possibility of tuning it to the

characteristics of the machine on which it is executed. This fact is the more advantageous since it often is a priori not clear what the grain size of a program should be in order to obtain a good performance.

Both techniques started from a fine grained solution. In the $[M,1]$-transformation we compose M cells of the fine grained solution resulting in a larger cell. In the $[K,K]$-solution we not only compose K cells of the fine grained program, but also compose larger messages by combining K communications into a single packet. In both techniques we are able to control the communication overhead of an implementation by setting the parameter to an appropriate value. In fact, both techniques are instances of a more general technique, the $[M,K]$-transformation, in which M cells are composed and in which K communications are composed into a single packet. It can be shown that given a fine grained program (of a certain regular structure) it is always possible to apply an $[M,K]$-transformation, provided that K is a divisor of M. Designing an application now boils down to two steps, viz. designing a fine grained program and applying an $[M,K]$-transformation [9]. By this approach we can benefit from the experience gained in the design of fine grained programs [10, 5, 4].

It is possible to implement our programs in VLSI [7]. In VLSI, chip area is one of the main design restrictions. We have seen that applying an $[M,1]$-transformation can reduce the total number of variables of a program and thereby reduce the chip area needed. Experiments with the DICY system at the Philips Research Laboratories have indeed demonstrated this property of the transformation.

We have carried out a number of experiments on a 51 T800-transputer network. For both the $[M,1]$-transformation and the $[K,K]$-transformation the experimental results are conform to the theoretical complexity analysis. In the experiments, each process is allocated to a separate processor. The computation will be more efficient if a number of processes share a processor, e.g. by a cyclic allocation scheme. In that case, however, the theoretical results presented in this paper do not apply anymore.

[1] B. Kruatrachue and T. Lewis, Grain size determination for parallel processing, IEEE Software, Jan. 1988, pp. 23–32.

[2] C. McCreary and H. Gill, Automatic determination of grain size for efficient parallel processing, Comm. of the ACM, Sept. 1989, Vol. 32, No. 9, pp. 1073–1078.

[3] D.J. Kuck, R.H. Kuhn, B. Leasure, and M. Wolfe, The structure of an advanced vectorizer for pipelined processors, Proc. 4th Int. Computer Software Appl. Conf. , Oct. 1980.

[4] M. Rem, Trace theory and systolic computations, in: J. W. de Bakker et al. , Eds. , PARLE Parallel Architectures and Languages Europe Vol. 1, Lecture Notes in Computer Science 258 (Springer-Verlag, Berlin, 1987) 14–33.

[5] A. Kaldewaij and M. Rem, The derivation of systolic computations, Science of Computer Programming 14 (1990), North-Holland, pp. 229–242.

[6] G. Zwaan, Parallel Computations, Ph. D. Thesis, Eindhoven University of Technology, The Netherlands (1989).

[7] K. van Berkel, J. Kessels, M. Roncken, R.W.J.J. Saeijs, and F. Schalij, The VLSI-programming language Tangram and its translation into handshake circuits, in: Proc. of the European Design Automation Conference, 1991.

[8] INMOS, Transputer instruction set: A compiler writer's guide, London, Prentice Hall, 1988.

[9] P. Struik, Designing parallel programs of parameterized grain size, POOMA document, no. 0111, Philips Research Laboratories Eindhoven, The Netherlands, Nov., 1989.

[10] P. Struik, A systematic design of a parallel program for Dirichlet convolution, Science of Computer Programming 15 (1990), special issue, North-Holland, pp. 185–200.

In this series appeared:

89/1	E.Zs.Lepoeter-Molnar	Reconstruction of a 3-D surface from its normal vectors.
89/2	R.H. Mak P.Struik	A systolic design for dynamic programming.
89/3	H.M.M. Ten Eikelder C. Hemerik	Some category theoretical properties related to a model for a polymorphic lambda-calculus.
89/4	J.Zwiers W.P. de Roever	Compositionality and modularity in process specification and design: A trace-state based approach.
89/5	Wei Chen T.Verhoeff J.T.Udding	Networks of Communicating Processes and their (De-)Composition.
89/6	T.Verhoeff	Characterizations of Delay-Insensitive Communication Protocols.
89/7	P.Struik	A systematic design of a parallel program for Dirichlet convolution.
89/8	E.H.L.Aarts A.E.Eiben K.M. van Hee	A general theory of genetic algorithms.
89/9	K.M. van Hee P.M.P. Rambags	Discrete event systems: Dynamic versus static topology.
89/10	S.Ramesh	A new efficient implementation of CSP with output guards.
89/11	S.Ramesh	Algebraic specification and implementation of infinite processes.
89/12	A.T.M.Aerts K.M. van Hee	A concise formal framework for data modeling.
89/13	A.T.M.Aerts K.M. van Hee M.W.H. Hesen	A program generator for simulated annealing problems.
89/14	H.C.Haesen	ELDA, data manipulatie taal.
89/15	J.S.C.P. van der Woude	Optimal segmentations.
89/16	A.T.M.Aerts K.M. van Hee	Towards a framework for comparing data models.
89/17	M.J. van Diepen K.M. van Hee	A formal semantics for Z and the link between Z and the relational algebra.

90/1	W.P.de Roever- H.Barringer- C.Courcoubetis-D.Gabbay R.Gerth-B.Jonsson-A.Pnueli M.Reed-J.Sifakis-J.Vytopil P.Wolper	Formal methods and tools for the development of distributed and real time systems, p. 17.
90/2	K.M. van Hee P.M.P. Rambags	Dynamic process creation in high-level Petri nets, pp. 19.
90/3	R. Gerth	Foundations of Compositional Program Refinement - safety properties - , p. 38.
90/4	A. Peeters	Decomposition of delay-insensitive circuits, p. 25.
90/5	J.A. Brzozowski J.C. Ebergen	On the delay-sensitivity of gate networks, p. 23.
90/6	A.J.J.M. Marcelis	Typed inference systems : a reference document, p. 17.
90/7	A.J.J.M. Marcelis	A logic for one-pass, one-attributed grammars, p. 14.
90/8	M.B. Josephs	Receptive Process Theory, p. 16.
90/9	A.T.M. Aerts P.M.E. De Bra K.M. van Hee	Combining the functional and the relational model, p. 15.
90/10	M.J. van Diepen K.M. van Hee	A formal semantics for Z and the link between Z and the relational algebra, p. 30. (Revised version of CSNotes 89/17).
90/11	P. America F.S. de Boer	A proof system for process creation, p. 84.
90/12	P.America F.S. de Boer	A proof theory for a sequential version of POOL, p. 110.
90/13	K.R. Apt F.S. de Boer E.R. Olderog	Proving termination of Parallel Programs, p. 7.
90/14	F.S. de Boer	A proof system for the language POOL, p. 70.
90/15	F.S. de Boer	Compositionality in the temporal logic of concurrent systems, p. 17.
90/16	F.S. de Boer C. Palamidessi	A fully abstract model for concurrent logic languages, p. p. 23.
90/17	F.S. de Boer C. Palamidessi	On the asynchronous nature of communication in logic languages: a fully abstract model based on sequences, p. 29.

90/18	J.Coenen E.v.d.Sluis E.v.d.Velden	Design and implementation aspects of remote procedure calls, p. 15.
90/19	M.M. de Brouwer P.A.C. Verkoulen	Two Case Studies in ExSpect, p. 24.
90/20	M.Rem	The Nature of Delay-Insensitive Computing, p.18.
90/21	K.M. van Hee P.A.C. Verkoulen	Data, Process and Behaviour Modelling in an integrated specification framework, p. 37.
91/01	D. Alstein	Dynamic Reconfiguration in Distributed Hard Real-Time Systems, p. 14.
91/02	R.P. Nederpelt H.C.M. de Swart	Implication. A survey of the different logical analyses "if...,then...", p. 26.
91/03	J.P. Katoen L.A.M. Schoenmakers	Parallel Programs for the Recognition of *P*-invariant Segments, p. 16.
91/04	E. v.d. Sluis A.F. v.d. Stappen	Performance Analysis of VLSI Programs, p. 31.
91/05	D. de Reus	An Implementation Model for GOOD, p. 18.
91/06	K.M. van Hee	SPECIFICATIEMETHODEN, een overzicht, p. 20.
91/07	E.Poll	CPO-models for second order lambda calculus with recursive types and subtyping, p. 49.
91/08	H. Schepers	Terminology and Paradigms for Fault Tolerance, p. 25.
91/09	W.M.P.v.d.Aalst	Interval Timed Petri Nets and their analysis, p.53.
91/10	R.C.Backhouse P.J. de Bruin P. Hoogendijk G. Malcolm E. Voermans J. v.d. Woude	POLYNOMIAL RELATORS, p. 52.
91/11	R.C. Backhouse P.J. de Bruin G.Malcolm E.Voermans J. van der Woude	Relational Catamorphism, p. 31.
91/12	E. van der Sluis	A parallel local search algorithm for the travelling salesman problem, p. 12.
91/13	F. Rietman	A note on Extensionality, p. 21.
91/14	P. Lemmens	The PDB Hypermedia Package. Why and how it was built, p. 63.

91/15	A.T.M. Aerts K.M. van Hee	Eldorado: Architecture of a Functional Database Management System, p. 19.
91/16	A.J.J.M. Marcelis	An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.
91/17	A.T.M. Aerts P.M.E. de Bra K.M. van Hee	Transforming Functional Database Schemes to Relational Representations, p. 21.
91/18	Rik van Geldrop	Transformational Query Solving, p. 35.
91/19	Erik Poll	Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.
91/20	A.E. Eiben R.V. Schuwer	Knowledge Base Systems, a Formal Model, p. 21.
91/21	J. Coenen W.-P. de Roever J.Zwiers	Assertional Data Reification Proofs: Survey and Perspective, p. 18.
91/22	G. Wolf	Schedule Management: an Object Oriented Approach, p. 26.
91/23	K.M. van Hee L.J. Somers M. Voorhoeve	Z and high level Petri nets, p. 16.
91/24	A.T.M. Aerts D. de Reus	Formal semantics for BRM with examples, p. 25.
91/25	P. Zhou J. Hooman R. Kuiper	A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and complete ness, p. 52.
91/26	P. de Bra G.J. Houben J. Paredaens	The GOOD based hypertext reference model, p. 12.
91/27	F. de Boer C. Palamidessi	Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.
91/28	F. de Boer	A compositional proof system for dynamic proces creation, p. 24.
91/29	H. Ten Eikelder R. van Geldrop	Correctness of Acceptor Schemes for Regular Languages, p. 31.
91/30	J.C.M. Baeten F.W. Vaandrager	An Algebra for Process Creation, p. 29.

91/31	H. ten Eikelder	Some algorithms to decide the equivalence of recursive types, p. 26.
91/32	P. Struik	Techniques for designing efficient parallel programs, p. 14.
91/33	W. v.d. Aalst	The modelling and analysis of queueing systems with QNM-ExSpect, p. 23.
91/34	J. Coenen	Specifying fault tolerant programs in deontic logic, p. 15.
91/35	F.S. de Boer J.W. Klop C. Palamidessi	Asynchronous communication in process algebra, p. 20.
92/01	J. Coenen J. Zwiers W.-P. de Roever	A note on compositional refinement, p. 27.
92/02	J. Coenen J. Hooman	A compositional semantics for fault tolerant real-time systems, p. 18.
92/03	J.C.M. Baeten J.A. Bergstra	Real space process algebra, p. 42.