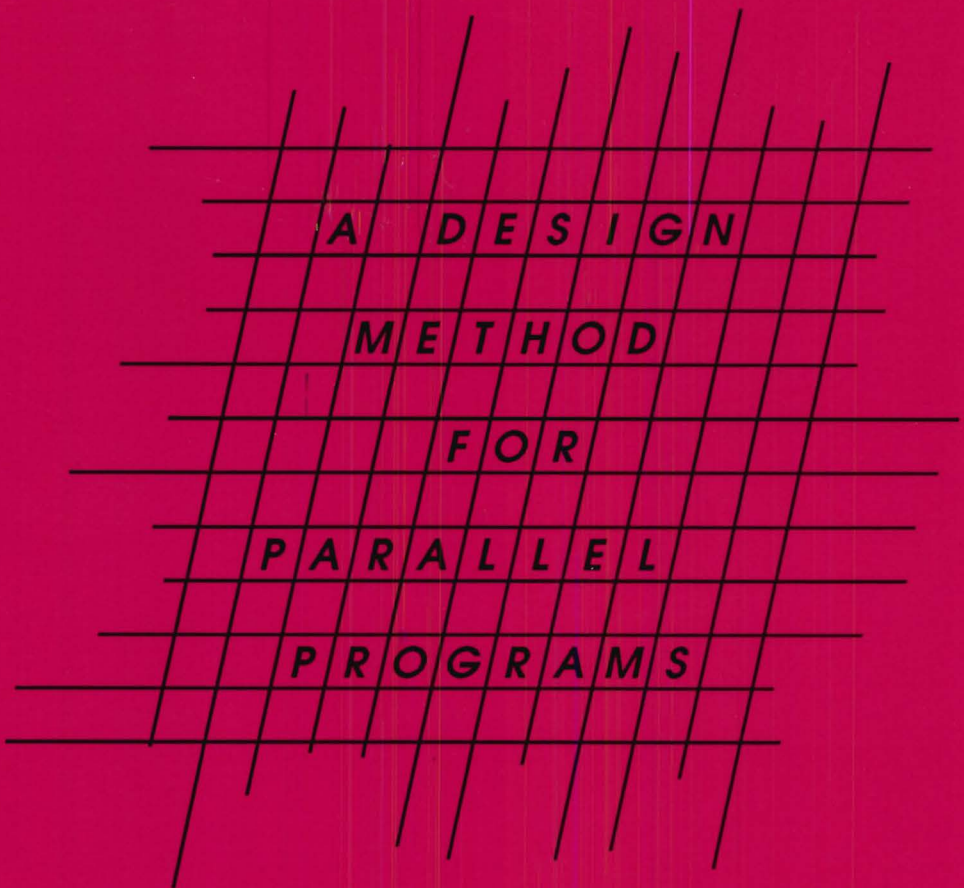
A design method for parallel programs



L. Daniël J.C. Loyens

A Design Method For Parallel Programs

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 aangewezen
door het College van Dekanen
in het openbaar te verdedigen
op dinsdag 22 september 1992 te 16.00 uur

door

Ludovicus Daniël Joseph Carolus Loyens

geboren te Maastricht

Dit proefschrift is goedgekeurd door
de promotoren

prof. dr. M. Rem
en
prof. dr. R.C. Backhouse

en de copromotor

dr. P.A.J. Hilbers.

The research reported in this thesis has been carried out at
Koninklijke/Shell-Laboratorium, Amsterdam
in collaboration with
Department of Mathematics and Computing Science,
Eindhoven University of Technology.

para Concha
e Quico

Acknowledgements

Writing a PhD thesis about parallel programs did not cross my mind until I met Hans van de Vorst. He and my former boss, Gerrit van Zee, convinced me to start working on the 'formal' part of KSLA's Parallel Computing project in 1987.

Many things changed, Hans en Gerrit moved and so did many of my ex-colleagues, but the Parallel Computing project kept on going and has been continuously supported by Rob Bisseling, John Somers and myself. Klaas Esselink and Peter Hilbers joined us, and carried over their enthusiasm into the project. Meanwhile the number of transputers of our parallel machine grew from one, to four, to forty, to four hundred.

I sincerely thank my colleagues for the many stimulating interactions during those exciting years. Special thanks go to Ben Sijtsma for his help with PostScript. Our managers Theo Verheggen and Arie Langeveld are gratefully acknowledged for providing me with the freedom to work on this thesis.

My main PhD adviser Martin Rem kept me focussed on the subject. It was a pleasure to work under his guidance. Thanks go also to the other members of the "kleine commissie": Roland Backhouse, Jan van Leeuwen and Bob Mattheij. Last but not least I thank Maria da Conceição Fernández Ferreira for her support and love.

Preface

Characteristic of today's scientific research is the use of digital computers as a modelling tool. For example, the effect of changing plant operations on the refinery process can be evaluated without on-site experiments. Another example is the simulation of fluid dynamics in a reactor.

A common characteristic of complex models is their huge demand on computational resources. Many problems are time and memory bound, and therefore require powerful computer systems. A diverse range of uni-processor systems is available, but it is generally felt that these uni-processor systems obstruct the realisation of larger models or faster solution times. Indeed, multi-processor systems are already used for many scientific computations.

Processing elements are more and more integrated on a single piece of silicon and allow for building system boards containing several processors. These system boards can be assembled to form a large multi-processor system. From a programming point of view, it desirable that such a system consists of well-balanced processing elements such as transputers.

The interactions between processors of such a system are carried out exclusively on the basis of message exchanging. The alternative is to have shared memory among the processors, but then the multi-processor system's size is bounded by the size of the shared memory. Distributed-memory systems overcome this barrier, since each processor has it own memory and interaction is via message passing. On the other hand, this kind of interaction can be very complex.

The use of multi-processor systems with a large distributed memory raises an important and practical question: "How do we program them?".

Perhaps this can be done by designing a compiler capable of extracting parallelism from a program. In our opinion, this is a tremendous task with many pitfalls. A more natural answer to this question is to design a parallel program for the target system. Such a program is difficult to design, since many complex program concerns have to be taken into account. Nevertheless, a strict design discipline like the one proposed in this thesis can be used to overcome this barrier. Our goal is to provide a formal method, which facilitates the design of parallel programs by seperating communication from computation issues.

Contents

Chapter 0

Introduction

The word parallelism originates from the Greek word $\pi\alpha\rho\acute{\alpha}\lambda\lambda\eta\lambda o\varsigma$. It is a contraction of $\pi\alpha\rho\alpha$, meaning side by side, and $\alpha\lambda\lambda\eta\lambda o\varsigma$, meaning one another. The spirit of this ancient Greek word breathes harmony (side by side) and similarity (one another).

A parallel program can be considered as a number of harmoniously cooperating processes with similar structures. The processes work towards a common goal and they usually interact with each other by exchanging messages. The processes we have in mind are imperative programs — prescriptions of statement sequences in some formal language.

A parallel program may involve many processes with complex interactions. It is generally felt that the high degree of complexity of a parallel program obstructs its development.

In this thesis, we discuss a design method for parallel programs. Our aim is to give some basic guidelines that enable us to master the complexity of parallel programs. This results in a parallel-programming method that is built on top of the formal methods for sequential program construction, which have been shown to be very succesful. Formal methods and sound engineering principles, like separation of concerns, are indispensable in parallel program construction. In the following, we informally explain our method.

Central in this thesis is the notion of a parameterised process. A parameterised process is much like a procedure or a subroutine in sequential programming. The difference is that, instead of having only one instantiation in a sequential program (a single call), we have many instantiations in a parallel program. Indeed, the parallel program is obtained by instantiating p processes from one single parameterised process. In this way, processes with similar structures are obtained.

There are more analogies with sequential programming. The design of a parameterised process closely resembles the design of a sequential program. A parallel program is specified using functional specifications just as in sequential programming. Such a specification forms the starting point for a parallel program derivation, i.e., a formal construction of parameterised processes constituting a parallel program. In

order to achieve this, we use parameterised invariants and other formal methods in the spirit of [13].

A parameterised process is further refined into a sequence of ordinary sequential programs[0] and communication processes. In this way, a parallel program is decomposed into *layers* of process instances.

Example 0.0

	$S.0$ ::	$S.1$ ::	$S.2$ ::
L_0	$S_0.0$	$S_0.1$	$S_0.2$
L_1	; $C_0.0$	; $C_0.1$	; $C_0.2$
L_2	; $S_1.0$	; $S_1.1$	; $S_1.2$
L_3	; $C_1.0$	; $C_1.1$	; $C_1.2$

Figure 0.0: Schematic of a parallel program decomposed into layers

In Figure 0.0, a parallel program consisting of 3 instances of parameterised process S is shown. Each process $S.q$, $0 \leq q < 3$, is vertically decomposed into a sequence of 4 processes, namely $S_0.q$, $C_0.q$, $S_1.q$, and $C_1.q$. Parameterised processes S_0 and S_1 are sequential programs. Parameterised processes C_0 and C_1 are communication processes.

Another way to look at the parallel program is by its decomposition into the layers: L_0 ; L_1 ; L_2 ; L_3. Each layer consists of instances of the same parameterised process. Communication can only take place between communication processes in the same layer, for example, between the processes $C_0.0$, $C_0.1$ and $C_0.2$ in layer L_1. □

Each *layer* consists of either communicating processes or cooperating programs, which are obtained from a parameterised process again.

This strict separation of concerns in the design of a parallel program has several advantages:

- A clear specification of each individual parameterised process can be given.
- The type of interaction between processes is limited, since a parameterised process is either an ordinary sequential program or a communication process. This facilitates the correctness proof of the parallel program.

The design of a communication process benefits from this approach, because its separate rôle becomes much clearer. For instance, it is relatively easy to analyse the

[0]We prefer to speak of sequential programs instead of sequential processes, since a process has always an interaction with its outside world. For the same reason, we prefer the word communication process above communication program. The entity that combines both a sequential program and a communication process is a process again.

influence of communication networks on the parallel program's complexity, and to consider alternative implementations for the communication processes.

Interaction between processes is either by message passing, which always takes place between processes within the same layer, or via local variables. The latter form of 'communication' crosses the boundary of the layers, and is similar to parameter passing in sequential programming. Each column in Figure 0.0 should be considered as one unit. An implementation of a parallel program on a multi-processor system simply means assigning each unit to a processor.

An efficient parallel program is guaranteed if we distribute the work and the number of communications evenly across the layers. A sequence of layers, L_0 ; L_1 ; L_2 ; ..., consisting of only sequential programs form a 'thick' computational layer. The work in such a layer should be distributed evenly across the parameterised processes of the parallel program. Cooperation, which addresses exchange or computations of global data, takes place in a communication layer. Preferably, such a layer should be 'thin', which can be achieved by avoiding communications as much as possible. If it is impossible to avoid communication we strive for spreading the communications across the instances in a communication layer. This clearly depends on the chosen communication network.

In this thesis several examples are given of parallel programs that are structured by their decomposition into layers. Included are non-trivial examples for segment problems, dense symmetric-system solving and sparse Cholesky factorisation. In each of the derivations we indicate the influence of the chosen communication network and the data distribution.

From our experience with parallel program construction on a medium-sized multi-processor system (400 transputers), we have found that the proposed method yields practical parallel programs, which can have high efficiencies [4].

Outline of the thesis

This thesis consists of seven chapters, which are numbered consecutively. The recommended order of reading is the sequence:

$$0\ ;\ 1\ ;\ 2\ ;\ \textbf{par}\ \ 3\ ,\ 4\ \textbf{rap}\ ;\ 5\ ;\ 6\ ,$$

where everything between **par** and **rap** can be read in arbitrary order.

Chapter 1 discusses several aspects of parallel programs. We state the skeleton of our design method, which is based on the use of parameterised invariants and our knowledge of sequential programming. Small examples are given to clarify our point of view. At the end of this chapter we give a derivation of a parallel program.

Chapter 2 discusses simple distributions of arrays and matrices together with two examples of making new distributions. The properties of distributions largely determine the efficiency of a parallel program. We study the properties of distributions in

order to obtain a better understanding of their rôle in parallel programs.

In Chapter 3, the proposed parallel programming techniques are applied to a class of segment problems. In this model problem, all ingredients of parallel program design are encountered. We start with deriving decomposition rules, and this eventually leads to the formulation of a parallel program scheme. The resulting techniques can also be applied to problems which closely resemble the model problem.

In Chapter 4, we derive a parallel program for the solution of a special class of symmetric systems. Parallel programs are given for the Cholesky factorisation and triangular system solving of a dense matrix. The parallel programs use a Cartesian distribution, which are very useful in matrix computations. It is demonstrated that it is feasible to strive for a separation between load balance and communication requirements. Some experimental comparison results are given as well.

Chapter 5 has a different character than the preceding chapters. A sparse parallel Cholesky factorisation program is obtained on the basis of the work of Chapter 4. The resulting algorithms are believed to be new, and can be used as building blocks for many algorithms that need the solution of a sparse symmetric positive-definite system. Timing-results for the parallel programs are also given.

Chapter 6 is the closing chapter of this thesis.

Notation

The notation for quantifications slightly differs from what is used in mathematics. The general format is:

$$(\odot\, k : Q : E)\ ,$$

where $\odot$ is a quantifier, for instance, Σ, $\mathbf{max}$, $\forall$, etc., k is a list of bound variables, Q is a predicate describing the domain of the bound variables, and E is an expression. The base type of the bound variables is usually the set of integers.

Sets are denoted in a similar way as quantifications. The notation

$$V = \{i, j : i^2 + j^2 = a^2 : (i, j)\}$$

specifies the set V of all integer pairs which lie on a circle around the origin with radius a. The cardinality of a set V is denoted by $\mid V \mid$.

Function application is denoted by a dot (.). It has the highest binding power and associates from right to left. Whenever confusion is possible a pair of parentheses has been added. The integer operations division and remainder use the symbols / and \, respectively.

The program derivations and proofs are recorded in the following notational style due to W.F.H. Feijen:

$$
\begin{array}{ll}
 & E0 \\
= & \quad \{ \text{ hint why } E0 = E1 \ \} \\
 & E1 \\
\geq & \quad \{ \text{ hint why } E1 \geq E2 \ \} \\
 & E2
\end{array}
$$

where Ei, $0 \leq i < 3$, are expressions. In this way, a derivation of $E0 \geq E2$ is recorded via an intermediate expression $E1$. A hint of the form $E0 = E1$ is an indication of how to obtain in a small number of steps the equality between expression $E0$ and $E1$. The hint "calculus" refers to common arithmetical rules.

Chapter 1

Design Methodology

The aim of this chapter is to describe in a nutshell a number of important concerns with respect to parallel programs. We briefly discuss in order of appearance: aspects of parallel program construction, program notation, proof rules, communication processes, time complexity, and the **parseq** rule. The discussion is tailored to our needs. Here, the main purpose is to set out the lines of thought for playing the game called parallel programming. To exemplify this game we give a derivation of a parallel program computing all partial combines.

1.0 Aspects of parallel program construction

1.0.0 Functional specifications

It is quite common in sequential programming to use the Hoare-triple [40]

$$\{Q\}\ S\ \{R\}$$

to denote a formal specification. This notation expresses that if program S starts in a state described by predicate Q and the program terminates, then upon completion predicate R is satisfied. Hoare-triples have been adequate in sequential program construction, and they can be extended to specify parallel programs as follows. Both pre- and postcondition, Q and R, are split up as the conjunction of p, $p > 0$, local pre- and postconditions, and a process is associated with each such pair. Specifically, the triple

$$\{Q.q\}\ S.q\ \{R.q\}$$

is the functional specification of process $S.q$, $0 \leq q < p$, where S is a parameterised process. In this way, only one single parameterised specification is given instead of p specifications.

A parameterised specification usually contains some local variables representing a part of a distributed data object, for example, an array or a graph. The processes of the parallel program perform operations on such a data object. Therefore, the parameters of a specification are q, p and a data distribution $\mathcal{D}$. Many choices are possible for $\mathcal{D}$, each of them having an impact on the complexity of the parallel program. It is assumed that variables representing distributed data are partitioned across the p processes. There is no shared memory.

For manipulating a parameterised specification, it is necessary to have the data distribution be parameterised as well. In Chapter 2 some data distributions are studied in more detail. An example of a parameterised specification is given next.

Example 1.0 (sum, specification) Given are p processes and an array f of length n distributed across all processes. The problem is to determine a parameterised process S that records in each process the sum of all array elements of f. The functional specification reads:

```
|[ p, n: int;
  f(i : 0 ≤ i < n): array of int;
  {0 < p ≤ n}
  par q : 0 ≤ q < p :
      |[ w: int;
        {Q.q : 0 < n}
        S.q
        {R.q : w = (∑ i : 0 ≤ i < n : f(i))}
      ]|
  rap
]| .
```

The parallel program is formed by p instances of S, namely all $S.q$, $0 \leq q < p$. In the notation, parallel composition is expressed by **par rap**. The brackets "|[" and "]|" are scope brackets and are used to delimit the extent (or scope) of a variable-declaration. Note that for $p = 1$ we have a specification of a sequential program. □

1.0.1 Invariants

The approach we follow to obtain a parameterised process S from a functional specification is similar to the methods used in sequential programming [13, 33]. These methods obtain from a specification an invariant in a calculational style. Several standard techniques are applicable to finding a suitable invariant. The programs are derived by calculating the necessary conditions to maintain the invariant. In a derivation, one often identifies subproblems that are easier to deal with than the original problem. This process of refinement is repeated until it becomes trivial to design a program text that meets its specification.

Our approach differs from others [9, 10, 70] in that the invariants are also parameterised as in [75]. This is a natural consequence of introducing parameterised specifications. There is no need to define a theory about parallel programs. Indeed, the main advantage of this approach is that we reuse sequential programming techniques.

In our approach, a programmer has to concentrate on a parameterised specification and has to obtain a parameterised invariant. This can be done, for example, by taking the data distribution into account. The aim is to rewrite the pre- and postcondition in such a way that one can identify local and global specifications. A local specification, referring to data that is local to a process, can be satisfied by a sequential program. A global specification requires some form of coordination between the processes, i.e., several processes have to interact with each other via message exchanging in order to satisfy the specification. It is exactly the latter concern that makes parallel programming difficult.

Example 1.1 (sum, outline) Take the specification of the previous example. Let $\mathcal{O}.q$ be the set of indices of array elements of f that are assigned to process q. As a first step towards an invariant, two subproblems are identified: recording the sum of all array elements locally in a variable v, and summing these accumulated values globally. This can be derived by rewriting the global sum in $R.q$ as:

$$
\begin{array}{ll}
 & (\sum i : 0 \le i < n : f(i)) \\
= & \quad \{ \text{ rewrite range } \} \\
 & (\sum q : 0 \le q < p : (\sum i : i \in \mathcal{O}.q : f(i))) \\
= & \quad \{ \text{ introduction } lsum \} \\
 & (\sum q : 0 \le q < p : lsum.q) \ ,
\end{array}
$$

where

$$lsum.q = (\sum i : i \in \mathcal{O}.q : f(i)) \ , \text{ for all } q : 0 \le q < p.$$

In this way, two subproblems $S0.q$ and $S1.q$, with local postconditions $R0.q$ and $R1.q$ respectively, are identified (cf. Figure 1.0).

$S.q$ ::
[[v: int;
  $S0.q$
  $\{R0.q : v = lsum.q\}$
  ; $S1.q$
  $\{R1.q : w = (\sum q : 0 \le q < p : lsum.q)\}$
]]

Figure 1.0: Outline of parameterised process S for the **sum** problem

For process $S0.q$ it is easy to obtain an invariant. Process $S1.q$ requires a global communication process, which will be discussed in Section 1.4. □

1.0.2 Correctness

The programs we intend to make should be correct by construction. So, correctness of parallel programs addresses correctness of applying the construction rules.

In non-communicating parts of a parameterised process we have only assertions and statement lists. The correctness can be proven by using assertions and proof rules based on the *wp* calculus [13, 15]. For instance, termination of loops, is proven in the usual way by a variant function which decreases in every iteration of the loop and is bounded from below.

Unfortunately, parallel programs interact with each other, thus complicating correctness rules considerably. The only interaction we allow is communication based on message passing. Processes send and receive values (messages) along channels. The part of a process instance that contains these communication statements is called a communication (sub) process.

Proof rules for communication statements are given in Section 1.2. An explicit assertion is made before the sending of a value, and an explicit assertion is made after the receipt of a value. The assertions about the communicated values are expressed in terms of global (constant) expressions. In this way interference of proofs, as encountered in the Owicki-Gries theory [64, 65], is avoided. This leaves us with the obligation to prove the correctness of a communication process using the proof rules. Such a proof in its full length can be quite cumbersome.

Usually the functionality of a communication process is a very simple one. It is sufficient, therefore, to study some frequently used communication processes and their implementation on communication networks (see Section 1.3). In this way, problems like deadlock and starvation are avoided, since it is assumed that correct implementations of communication processes can be given.

The communication processes are parameterised as well, and they can be specified in isolation. This not only facilitates the correctness proof, but also allows for a single correctness proof. In addition to this, a strong restriction is posed on the process instances of a parameterised communication process. All communications occur between instances of the same parameterised process. Such a process is called communication closed. Therefore, one can think of a parallel program as being decomposed into *layers*. Each layer either is a sequential statement list or contains communication statements. Layers can be syntactically separated by semicolons, and are specified by pre- and postconditions.

Example 1.2 A parameterised process S might be decomposed into

$$
\begin{array}{|l}
\hline
S.q :: \\
\quad S0.q \\
; \; C0.q \\
; \; S1.q \\
\hline
\end{array}
$$

where $S0.q$ and $S1.q$ are sequential programs and $C0.q$ is a communication pro-

cess. $S0.q$ may be a complex program; the process instances of $S0$ form a layer and have parameterised preconditions and postconditions. The communicating processes instances $C0.q$ also form a layer. It is possible that each process $S1.q$ is further decomposed into, for example:

```
S1.q ::
  do B.q
  →  S2.q
     ; C1.q
     ; S3.q
  od
```

where $B.q$ is a guard (a boolean expression in terms of local variables of process q), $S2.q$ and $S3.q$ are sequential programs, and $C1.q$ is a communication process. In the **do**-loop the layers formed by the processes $S2.q$, $C1.q$, and $S3.q$, are identified. It is possible that each process instance $S1.q$ is executing in a different layer, but *logically* $S1$ is decomposed into layers whose correctness proofs are given separately. The correctness of the loop in $S1$ is partially proven by:

$$\{P.q \wedge B.q\}\ S2.q\ ;\ C1.q\ ;\ S3.q\ \{P.q\}$$

for an invariant $P.q$ of $S1.q$. □

The term layer in the context of parallel programs was first introduced in [20]. In their terminology a layer of mutually communicating processes is called a communication-closed layer. In [20] the layers were only used to verify correctness of parallel programs. In this monograph we use the *layer* concept as part of the design methodology. The programmer is, of course, responsible for the formulation of logical layers.

So much for the design aspects of parallel programs. In the sequel we shall omit the parameters p and $\mathcal{D}$ in parameterised formulas (specifications, invariants), and the range $0 \leq q < p$ for process numbers.

1.1 Program notation

The program notation used is based on Dijkstra's guarded command language and is described in [13, 15]. Examples of sequential programs written in the guarded command language can be found in [14, 46].

In the programs we have declarations of variables in a Pascal-like style, extended with local scope rules. The symbols $[\![\ x \ldots\ ]\!]$ delimits the scope of variable x. Variables have usually type 'int' or 'real'. An array f of length n with base type 'real' is declared as "$f(i : 0 \leq i < n)$: array of real". The programs have the following constructs:

abort	stop forever
skip	do nothing
$x := e$	assignment
$S0\ ;\ S1$	sequential composition
if $B0 \rightarrow S0 \parallel B1 \rightarrow S1$ **fi**	alternative construct
do $B0 \rightarrow S0 \parallel B1 \rightarrow S1$ **od**	repetition.

The **if** and **do**-statements use boolean expressions $B0$ and $B1$, called guards. If any guard in an **if**-statement evaluates to *true* the corresponding alternative is chosen; if all evaluate to *false* the statement is equivalent to *skip*. In a **do**-statement guards evaluating to *true* and their corresponding alternatives are chosen repeatedly; if all evaluate to *false* the statement is equivalent to *skip*. We allow more than two alternatives.

As an extension to Dijkstra's notation we introduce the **for all**-statement:

$$\textbf{for all}\ i : i \in set : S.i\ \textbf{lla rof} \qquad \text{arbitrary order}.$$

This statement denotes sequential composition in some arbitrary order of statements $S.i$, one for each value i in the set. If the set is empty the **for all**-statement is equivalent to *skip*. Note that the variable i in the range of **for all** is a bound variable. Sometimes more than one bound variable appears in a range, thus specifying a nested repetition.

Example 1.3 (sum, $S0.q$) The statement list

$$\begin{array}{|l} v := 0 \\ ;\ \textbf{for all}\ i : i \in \mathcal{O}.q : v := v + f(i)\ \textbf{lla rof} \end{array}$$

assigns to v the sum of all array elements f that are local in process q. ($\mathcal{O}.q$ gives for each process q the set of local array indices of f.) This statement list implements process $S0.q$ of Example 1.1. □

Parallel composition is denoted by

$$\textbf{par}\ q : 0 \leq q < p : S.q\ \textbf{rap} \qquad \text{parallel composition}.$$

We allow different ranges in the **par**-statement and different process identifications. The **par**-statement terminates if all of its constituent processes terminate.

Example 1.4 The statement list

$$\textbf{par}\ s,t : 0 \leq s < M \wedge 0 \leq t < N : S.s.t\ \textbf{rap}\ ,$$

with $p = M * N$, specifies the parallel composition of p processes $S.s.t$ each identified by an ordered pair (s,t), $0 \leq s < M$ and $0 \leq t < N$. □

The notation

$$\textbf{par } S0 \ , \ S1 \textbf{ rap}$$

denotes the parallel composition of two processes $S0$ and $S1$, and is used in communication processes to express simultaneous execution.

Communication is expressed by the statements:

$r!e$	output to process r the value of expression e
$s?x$	input from process s of a value, which is assigned to x.

Since all processes are identified by process numbers, two processes r and s performing $s :: r!$ and $r :: s?$ define a channel in the CSP meaning [42], namely from process s to process r. Such a channel is shared between two processes, and its direction is from sender to receiver. Every output statement is matched by a unique input statement and vice versa. We have not used names to denote channels between processes, since it is always clear in the programs which process is sending (outputting) and which process is receiving (inputting). Process numbers can be used in expressions in order to identify a channel.

Example 1.5 The statement

$$(q+1)!10 \ , \text{ for a process } q \ ,$$

denotes that process q sends the value 10 to process $q+1$. The expression $(q+1)$ is called a channel expression. □

The underlying communication mechanism can be synchronous (like in CSP) or asynchronous. For the latter it is required that messages (values) sent by a process always arrive, in arbitrary time, at the receiving process, without duplication, and in the same order they were sent.

The parallel programs presented in this monograph do not use communication channels, like a *probe* [61] or the ALT construct of occam [43], in guards. Such a construct is difficult to capture in a simple proof rule for communication.

1.2 Proof rules

A *weak* correctness proof of a parallel program, i.e., in the absence of deadlock, relies on the annotated program and the use of proof rules. An annotated program uses assertions before and after the program (corresponding to the functional specification), and between its statements. Such an assertion, or Hoare-triple,

$$\{Q\} \ S \ \{R\} \ ,$$

is valid if it is either an axiom, or it is obtained by applying an inference rule.

An example of an axiom is the assignment axiom:

$$\{R\ (x := e)\}\ x := e\ \{R\}\ ,$$

The notation $R\ (x := e)$ refers to predicate R with all free occurrences of x replaced by expression e.

An inference rule of the form:

$$\frac{A,\ B}{C}$$

where A, B, and C are predicates, states that if A and B are proved then C may be concluded. An example of an inference rule is the rule of consequence from sequential programming:

$$\frac{Q \Rightarrow Q',\ \{Q'\}\ S\ \{R'\},\ R' \Rightarrow R}{\{Q\}\ S\ \{R\}}\ ,$$

stating that a precondition may always be strengthened and a postcondition may be weakened.

The axioms and inference rules form the set of proof rules. The parallel programs we consider all terminate, and consist of sequential statements taken from Dijkstra's language and statements for expressing parallelism and communication. For the sequential statements the proof rules from [13, 33] are used. It is assumed that the corresponding axioms and inference rules for the program constructs like ; , $x := e$, etc., are known.

Additionally, we have the following proof rule for the **par**-statement [41, 65]:

Definition 1.6 (par rule)

$$\frac{(\forall\, i : 0 \le i < p : \{Q.i\}\ S.i\ \{R.i\})}{\{(\forall\, i : 0 \le i < p : Q.i)\}\ \textbf{par}\ q : 0 \le q < p : S.q\ \textbf{rap}\ \{(\forall\, i : 0 \le i < p : R.i)\}}$$

□

The precondition of the **par**-statement is the conjunction of all preconditions of each process $S.q$. A similar remark holds for the postcondition. The state spaces of the processes are disjoint, since each process has its own set of local variables. Execution of one process, therefore, cannot alter the state of another process, except when communication occurs. Communication is done by message passing; proof rules are given elsewhere.

The **par** rule states that if we manage to design a parameterised process $S.q$, with corresponding pre- and postcondition, then we have designed the parallel program.

A derivation of a parallel program can be started in two different ways. The most natural way is to start from a local postcondition and closing the gap between pre- and postcondition by refinement. Sequential programming techniques can be used for

refining. Another way is the formulation of global invariants, i.e., an invariant about the distributed data object as a whole. Global invariants serve as an intermediate step towards local invariants. We will see an example of the latter in Section 1.5.

The input statement uses the following axiom:

Definition 1.7 (input axiom)

$$r :: \{true\}\ s?x\ \{M.x\}\ ,$$

where $M.x$ is a predicate in terms of local variable x of process r and its process number. s is the process number of the sending process. □

The axiom is adapted from [51] and states that anything can be concluded after the receipt of a value. At first sight, this may be a rather strong conclusion, but if we consider the input axiom in isolation then, with only one process running, an input statement deadlocks and any predicate $M.x$ may be assumed to be true upon termination.

The next inference rule is similar to the rule of satisfaction in [51]. It relates the postcondition of an input statement in a process r to the precondition of the matching output statement in process s.

Definition 1.8 (?!-rule)

$$\frac{r :: \{true\}\ s?x\ \{M.x\}}{s :: \{M.x\,(x := e)\}\ r!e\ \{M.x\,(x := e)\}}$$

where $M.x$ is a predicate in terms of the local variable x of process r and its process number, and e is a local expression of process s. Furthermore, the communication statements in s and r match. □

The motivation for the ?!-rule is the following. In the design of a communication process we have available the postcondition of a receiving process r, and in particular, an assertion $M.x$ about the, to be, communicated value x. The precondition of the matching sending process s is easily obtained by substitution. The sending of a value does not change the state space of the sending process s. Unlike in [51], no assertions are made about the global state space. Indeed, the proof rule presented here is rather weak, since it makes only assertions about the communicated value. For our purposes, however, it suffices, and it is also applicable to asynchronous communication.

Example 1.9 An application of the ?!-rule is given next. Take $M.x \equiv odd.x$ and $e = y + 1$ in Definition 1.8. The precondition of the sending process becomes:

$$\begin{array}{ll} & M.x\,(x := e) \\ \equiv & \quad \{\ \text{definitions } M.x,\ e\ \} \\ & odd.x\,(x := y + 1) \\ \equiv & \quad \{\ \text{substitution}\ \} \\ & odd.(y + 1) \\ \Leftarrow & \quad \{\ \text{calculus}\ \} \\ & even.y\,. \end{array}$$

From this example we see that communication is like a distributed assignment $x := e$. The right-hand side of $x := e$ is evaluated by process s, and the result is communicated to process r. Process r assigns the received message to variable x. Note that x is a local variable of process r. □

It is allowed to have process numbers in the expressions e and $M.x$.

Example 1.10 Consider a parallel program with two parameterised processes $S.q$, $0 \leq q < 2$, i.e., **par** $S.0$, $S.1$ **rap**, and

$S.q$ ::
‖[x: int;
  $\{0 \leq q < 2 \wedge B.q\}$
  par $\{B.q\}\ (1-q)!q\ \{B.q\}$, $\{B.q\}\ (1-q)?x\ \{M.x.q\}$ **rap**
  $\{B.q \wedge M.x.q\}$
]| .

Figure 1.1: Outline of two parallel processes $S.q$ with $q = 0$ or $q = 1$

In the assertions, predicate $M.x.q$ equals $x = 1 - q$. To use the communication rules we have to identify the matching communications. In this case, there are two matching communications, namely

$$\begin{array}{rl} 1-q :: & \{B.(1-q)\}\ q?x\ \{M.x.(1-q)\} \\ q :: & \{B.q\}\ (1-q)!q\ \{B.q\}\ . \end{array}$$

Applying the ?!-rule for $r = 1 - q$ and $s = q$, $0 \leq q < 2$, we obtain for $B.q$ the following parameterised proof:

$$\begin{array}{ll} & M.x.(1-q)\,(x := e) \\ \equiv & \quad \{\ \text{definitions } M.x.(1-q),\ e\ \} \\ & x = q\,(x := q) \\ \equiv & \quad \{\ \text{substitution}\ \} \\ & \mathit{true} \\ \equiv & \\ & B.q\ . \end{array}$$

Hence, the precondition is *true*. □

The communication rules are not applied as strictly as suggested. In general, we indicate their use in the correctness proof of a communication process. The assertions in communication processes are annotated according to the ?!-rule. This means that a postcondition is given for the receiving process after an input statement (?), and a precondition is given for the sending process just before its output statement (!).

1.3 Communication processes

Consider the following specification of a parameterised communication process.

Example 1.11 (broadcast) Let s be a process number and X be a value.

$$\boxed{\begin{array}{l}\{q \neq s \vee x = X\} \\ C.q \\ \{x = X\}\end{array}}$$

In C, a value X from process s is sent to all other processes. Such a functionality is called a *broadcast*. Variations are multiple senders (multi broadcast) or only a subset of the processes as receivers. □

The specification of a communication process like a broadcast is relatively simple, its implementation is not. The reason is that additional assumptions have to be made about the channels that are used during the broadcast. This restriction stems from the physical limitations a processor network has. In an implementation, processes are assigned to processors, and channels to paths of communication wires connecting processors. Needless to say that this burdens the programmer, and complicates the task of algorithm design, since size and processor topology have to be taken into account.

A parallel program can be considered [60] as an undirected graph with processes as nodes. Each edge represents two communication channels of opposite direction (an edge indicates the possibility to communicate directly between two processes). Additionally, it is often required that such a "computation graph" is connected and may vary in structure during the computation. On the other hand, a processor network can be considered as a *fixed* "implementation graph" representing a multiprocessor system. Mappings between these graphs have been studied by others [7, 39], and in this monograph we do not consider the mapping problem. Instead, we briefly discuss some simple communication networks [77], and forget about processor networks.

It is possible to write communication processes independent of any process topology by using spanning trees as demonstrated in [58]. Still, it is desirable to consider different kinds of communication networks, since they have a certain impact on the time complexity of the parallel program. Isolated specifications for communication processes allow for alternative realisations. In this way, it is possible to analyse the influence of communication networks on the communication processes.

1.3.0 Some simple communication networks

The simplest *connected* communication network one can think of is a tree of p nodes. The degree of a node in a network (graph) is the number of neighbours of the node. Trees with nodes of degree at most 2 can be obtained be arranging nodes as a *chain*,

also called linear array. In a *chain* network of p processes, two processes s and t, $0 \leq s, t < p$, are neighbours iff $\mid s - t \mid = 1$.

An important measure for the number of communication steps is the length of a longest path, i.e., the maximal number of edges on a path. A chain has a longest path of length $p - 1$. A tree with shortest-possible longest path is the star network of p nodes. The path length is at most 2: one node has degree $p - 1$ and the other nodes [0] (if present) have degree 1 .

A balanced binary tree (every non-leaf has two neighbours) consists of $2^k - 1$ nodes, $k > 0$, of which 2^{k-1} are leaves. A balanced binary tree has a longest path to the root of length $k - 1$.

A simple cyclic communication network is the *ring* of p nodes, which can be obtained by adding an edge between nodes 0 and $p - 1$ in the *chain* network. The length of a longest path in a ring is $p/2$.

New communication networks can be obtained by taking the Cartesian product of graphs [36]. For example, an M by N *mesh*, $M, N > 0$, is the Cartesian product of two *chains* of length M and N. A *torus* or *toroid* is the Cartesian product of two *ring* networks.

A *binary hypercube* of dimension n has 2^n vertices and is obtained by the Cartesian product of n *chains* of length 2. In general, the *k-ary n-cube* (the case $k = 2$ is usually called *hypercube*), is characterised by k^n vertices, each vertex having an n-digit radix-k address and an edge iff two vertices differ in their addresses by 1 in only one digit.

Many of the simple communication networks mentioned before can be emulated on a binary hypercube [7].

Finally, we have the complete network in which every two processes can communicate directly by their channel.

1.4 Time complexity of parallel programs

The time complexity of a parallel program is an estimate of the time it takes to execution the program's computation. It is desirable to quantify the time complexity of a parallel program in a mathematical formula. Such a formula relates the execution time of a parallel program to the size of the input n and the number of processes p.

A process can compute, can communicate, or, in the absence of these main activities, can be idle. Each activity contributes to the time complexity of a parallel program. We introduce two measures: the computation complexity and the communication complexity, each including (a part of) idling. The rôle of each of these complexities will be discussed in turn. We start with the latter.

[0] A node of degree 1 in a tree is called a leaf.

The communication complexity, denoted by $T_c.p.n$, is determined by counting the number of communication steps needed to complete the communication process. A communication step is the communication of a value to a neighbouring process. The communication network can allow for a number of communication steps to take place in parallel. This means that $T_c.p.n$ is at most the total number of communications.

The communication complexity is also determined by the number of values (messages) to communicate and the size of a message. We assume that all messages have uniform size.

Example 1.12 (broadcast) A *broadcast* can be implemented on a *chain* according to Figure 1.2.

$C.q ::$
$\{0 \leq s < p\}$
$\{q \neq s \vee x = X\}$
if $q < s \rightarrow (q+1)?x \;[]\; s < q \rightarrow (q-1)?x$ **fi** $\{x = X\}$
; **par** **if** $0 < q \leq s \rightarrow (q-1)!x$ **fi**
, **if** $s \leq q < p-1 \rightarrow (q+1)!x$ **fi**
rap $\{x = X\}$

Figure 1.2: Implementation of a broadcast on a chain

The correctness of this communication process can be proven by induction on the number of processes p and using proof rules.

The number of communication steps needed to complete the broadcast for a *chain* is at most $p-1$, since the longest path has length $p-1$ for $s = 0$. Clearly, this is a worst-case scenario; the average path length is:

$$\frac{1}{p} * (\sum\, s : 0 \leq s < p : s \;\mathbf{max}\; (p-1-s)) \;,$$

which is approximately $(3{*}p)/4$, and this can also be achieved without the parallelism in the sending process $s = q$. Note that in a broadcast process where a value is sent from one process to only one other process the average path length becomes approximately $p/3$. A broadcast process on a hypercube takes $\log p$ steps. □

Example 1.13 (sum, $S1.q$) Consider the specification of communication process $S1.q$ for the **sum** problem (see Example 1.1). It is necessary to add all local *lsum* values. This can be done by using a tree in which these values are collected and added from the leaves towards the root. The root, process 0, will broadcast the global sum to all processes. We use a tree with minimal longest-path length.

For each process q we define the functions $father.q$ and $children.q$ by:

$$\begin{aligned} father.q &= (q-1)/2 \text{ if } q > 0 \text{ otherwise } 0, \\ children.q &= \{i : (i = 2*q+1 \vee i = 2*q+2) \wedge i < p : i\} \;. \end{aligned}$$

Process 0 is the root, i.e., $father.q = q$ holds for $q = 0$. The program text for $S1.q$ is given in Figure 1.3. In the assertions we use function $Tree$. $Tree.r$ gives the set of all processes in the subtree rooted at process r. A formal definition is:

$$Tree.r = \{r\} \vee (\cup\, k : k \in children.r : Tree.k) \ .$$

For simplicity reasons, an array x is declared in every process recording intermediate results. (In a actual implementation each process needs at most two variables.)

The communication complexity $T_c.p.n$ of this process is $\mathbf{O}(\log p)$, since we use a tree with minimal longest-path length $\lfloor \log_2 p \rfloor$. Had we chosen to implement process $S1.q$ on a *chain* network then the complexity would become $\mathbf{O}(p)$. □

```
S1.q ::
[[ x(0 ≤ i < p − 1): array of int;
    par r : r ∈ children.q :
          r ? x(r)
          {x(r) = (∑ k : k ∈ Tree.r : lsum.k)}
    rap
  ; w := v
  {w = lsum.q}
  ; for all r : r ∈ children.q : w := w + x(r) lla rof
  {w = (∑ k : k ∈ Tree.q : lsum.k)}
  ; if father.q ≠ q →  father.q ! w
                     ; father.q ? w
                       {w = (∑ q : 0 ≤ q < p : lsum.q)}
    fi
  {w = (∑ q : 0 ≤ q < p : lsum.q)}
  ; for all r : r ∈ children.q : r ! w lla rof
]]
```

Figure 1.3: Outline of each communication process $S1.q$ for the **sum** problem. The communication network used is a tree.

From the last two examples we learn that the chosen communication network heavily influences the time complexity of communication processes, and hence the overall time complexity. This is not always the case, since it can happen that the communication complexity is entirely determined by the number of messages, or is dominated by the amount of work per process.

Example 1.14 (pipe-lining) Given is a broadcast process that instead of one value broadcasts n values. Such a situation happens when it is necessary to broadcast an array of n elements. We assume that n is much larger than p ($n \gg p > 1$). The time complexity of such a broadcast process on a *chain* network is

$$T_c.p.n = n + p - 2$$

using a technique called pipe-lining.

The program is obtained by repeatedly performing, for each array element, the broadcast program C (cf. Figure 1.2).

The time complexity consists of two terms. The first term indicates the number of communications n. The second term is recovered for $n = 1$ and indicates the length $p - 1$ of the pipe, which is at most the length of the longest path in the communication network. This is usually called the start-up time of the pipe, since it equals the number of steps to send a message along the pipe. For large $n \gg p$, $T_c.p.n$ is approximately n on any communication network.

The communication complexity of this problem is entirely determined by the number of values to communicate, and not by the communication network. □

Example 1.15 The communication complexity becomes less important when so-called surface/volume effects play a rôle. Imagine a computation on the pixels of a two-dimensional picture [21], where each pixel uses information from all its nearest-neighbour pixels.

The picture can be represented by an n by n matrix of values, one for each pixel. A natural way of distributing a square matrix is by using a $\sqrt{p}$ by $\sqrt{p}$ mesh of processes (assuming p is square). Each process in the mesh is responsible for a pixel submatrix.

This leads to a parallel program with communication complexity $\mathbf{O}(\frac{n}{\sqrt{p}})$, since it is determined by number of the pixels on the borders of each submatrix. The amount of work per process is $\mathbf{O}(\frac{n^2}{p})$, since computations are done for all pixels in a submatrix.

It is clear that for n large compared to $\sqrt{p}$ the amount of work per process dominates the communication complexity. The network hardly influences the total program's complexity. □

The computation complexity of a sequential program is usually determined by counting the number of elementary operations (assuming every elementary operation takes the same amount of time and ignoring overhead). In the parallel case this is not an adequate model, since operations can overlap.

As previously motivated, we construct our programs in *logical layers* and each layer is either a sequential program or a communication process. Synchronisation is enforced naturally between a sequential program and a communication process if we assume synchronous communication.

The time complexity of a sequential program layer is determined by the slowest process instance in that layer. Usually, the slowest process has to perform most operations. Therefore, it is meaningful to count operations and to compare the minimum number of operations in a process with the maximal number. The time complexity of a communicating layer is determined by the communication complexity.

The time complexity of the parallel program is obtained by summing the time complexities of each layer. In this way, we obtain an upper bound on the time complexity of the parallel program. If we ensure (and we will!) that the program layers are well balanced then this upper bound is a good estimate for the actual time complexity.

There is, however, a problem with asynchronous communication which allows for overlapping of computations and communications within a process. This combination is difficult to capture in complexity results. For this reason, the complexity results are only valid for synchronously communicating processes.

We wish to relate the communication and the computation complexity, and express the time complexity of a parallel program in formulas. For that purpose we define the quantity α.

Definition 1.16 (α, t_c, t_f) The communication-to-computation ratio α is defined by:

$$\alpha = \frac{t_c}{t_f},$$

where t_f is the time required to do a single elementary operation (an addition or multiplication), and t_c is the time required to communicate a single value of fixed size (an integer or a real). □

The values for α range from 0.5 for VLSI [2], about 4.5 for transputers [4], to about 150–750 for the current generation hypercube machines [18]. The time complexity of a parallel program is given by $T.p.n$.

Summarising:

Definition 1.17 $(T.p.n, T_f.p.n, T_c.p.n)$

$$\begin{aligned} T.p.n &= T_f.p.n + \alpha * T_c.p.n\,, \\ T_f.p.n &= \text{the computation complexity}\,, \\ T_c.p.n &= \text{the communication complexity}\,. \end{aligned}$$

The communication complexity is obtained assuming *synchronous* communication. □

Note that the actual time spent by the parallel program is at most $T.p.n * t_f$. Sometimes we say that the time complexity is $\mathbf{O}(x)$ meaning $T.p.n = \mathbf{O}(x)$ where $\mathbf{O}$ denotes Landau's O symbol.

Example 1.18 (sum, complexity) In the sum problem, two layers are identified. The first layer is formed by all process instances $S.q$, which are sequential programs. The second layer is formed by the communicating instances $S1.q$.

The time complexity of $S0.q$ is:

$$T_f.p.n = (\mathbf{max}\ q : 0 \leq q < p : \mid \mathcal{O}.q \mid -1)\,.$$

This suggests to distribute the n array elements evenly across p processes, and hence

$$T_f.p.n = (n + p - 1)/p - 1\,.$$

The communication complexity of $S1.q$ is:

$$T_c.p.n = 2 * (\alpha^{-1} + 1) * l\,.$$

where l is the length of the longest path (to the root) in the tree, i.e., $l = \lfloor \log_2 p \rfloor$. Explanation: each non-leaf performs at most two additions (its local *lsum* plus two received values) and communication takes two α time units (for simplicity, we charge α time units for a parallel communication).

The time complexity of the parallel program for the **sum** problem on a balanced tree with longest path l becomes:

$$T.p.n = (n + p - 1)/p + 2 * (\alpha + 1) * l - 1 \ .$$

Note that this formula is correct for $p = 1$ ($l = 0$). □

Two other important notions in parallel time complexity are *speed-up* and *efficiency.*

Definition 1.19 (speed-up) The speed-up $S.p.n$ of a parallel program on a problem X of size n is defined by

$$S.p.n = \frac{T_{\text{seq}}.n}{T.p.n} \ ,$$

where $T_{\text{seq}}.n$ denotes the complexity of the *best* known sequential algorithm for problem X on the same platform. □

Speed-up is a measure for comparing the solution time of a certain problem with a fixed size solved in two different ways (as opposed to [35]). The aim is to be able to state how a parallel algorithm performs compared to its sequential brother. It is not always an easy task to obtain complexity formulas for parallel and sequential algorithms. Therefore, it is sometimes necessary to verify the parallel time complexity and speed-ups experimentally.

A fair comparison requires that both the sequential and the parallel algorithm use the same platform, i.e., identical software and hardware. Such experiments are in practice often impossible due to the absence of efficient *sequential* and *parallel* compilers. Instead the sequential program obtained from the parallel program for the case $p = 1$ is used, and $T_{\text{seq}}.n$ is approximated by $T_{\text{f}}.1.n$. Therefore, speed-up results should always be interpreted with a critical eye.

Related to speed-up is efficiency $E.p.n$, a measure for the degree of utilisation of the processes of a parallel program.

Definition 1.20 (efficiency) The efficiency $E.p.n$ is defined by:

$$E.p.n = \frac{S.p.n}{p} \ .$$

□

Efficiency is a number between 0 and 1 if we admit that the speed-up $S.p.n$ is at most p.

Example 1.21 (sum, efficiency) From the time complexity $T.p.n$ for the **sum** problem, and the sequential time complexity $T_{\text{seq}}.n = n - 1$, we can compute the efficiency of the parallel program.

$$E.p.n = (1 + \frac{2 * (\alpha + 1) * l * p - (n + p - 1) \backslash p}{n - 1})^{-1} \ , \text{ for } n > 1 \ ,$$

where $l = \lfloor \log_2 p \rfloor$.

From the last formula we learn that an efficiency of at least 50 % is guaranteed when the problem size per process ($\approx \frac{n}{p}$) is greater than $2 * (\alpha + 1) * l$. □

It is, in general, difficult to obtain high efficiencies for a large range of values for p. Usually, the lower order terms in the complexity results cannot be neglected for small values of p. Blowing up the problem size while fixing the number of processes often yields high efficiencies, but this approach is not always feasible for practical problems.

1.5 Combines and partial combines

As a first example of the use of parameterised invariants we give a derivation of a *combine* process. It often happens that in a parallel program a global sum or a global maximum of p values has to be computed. Such an operation is called a combine. By grouping the different values in a tree-like fashion (recursive doubling) one can efficiently compute such a combine [49]. Sometimes it is required to compute all partial combines[1], for example, all begin sums [12]. A specification of the latter is as follows:

```
|[ k, p: int;
  f(i : 0 ≤ i < p): array of int;
  {0 ≤ k ∧ p = 2^k}
  par q : 0 ≤ q < p :
      |[ m: int;
        S.q
        {R.q : m = M.0.(q + 1)}
      ]|
  rap
]| .
```

where

$$M.a.b = (\odot\, i : a \le i < b : f(i)) .$$

We assume that integer array f is distributed by assigning element $f(q)$ to process q (the identity distribution). The operator $\odot$ is associative, so any term in $M.a.b$, $a < b$, may be split off. Examples of $\odot$ are:

$$\begin{aligned} x \odot y &= x + y && \text{global sum} \\ x \odot y &= x \ \mathbf{max}\ y && \text{global maximum} . \end{aligned}$$

A variation of this problem is, for example, all partial end combines $M.q.p$. Note that value $M.0.p$ is the global combine.

[1] This problem is also known as the parallel prefix problem [50].

Our aim is to give a derivation of a partial combine process which has logarithmic time complexity. We start the derivation by first obtaining a global postcondition R' from the local ones.

$$R' \ : \ (\forall\, q : 0 \leq q < p : m_q = M.0.(q+1))\,.$$

The notation m_q refers to variable m of process q. The hidden constant k in $p = 2^k$ suggests an induction, and by replacing k by a variable t we obtain a global invariant P'.

$$P' \ : \ (\forall\, q : 0 \leq q < 2^t : m_q = M.0.(q+1)) \wedge 0 \leq t \leq k\,.$$

Here, we assume that t is global for all processes.

Note that P' is easily satisfied if we set m of process 0 to $f(0)$ and $t = 0$. Furthermore, P' implies the postcondition if $t = k$. Progress is made by increasing t, and t can be seen as a global clock. From P' we get the local invariants $P.q$.

$$\begin{aligned} P.q &: P0.q \wedge P1.q \\ P0.q &: 0 \leq t \leq k \\ P1.q &: 0 \leq q < 2^t \Rightarrow m = M.0.(q+1)\,. \end{aligned}$$

Via a global postcondition we ended with a local invariant. We also could have obtained the local invariant $P.q$ directly by introducing variable t immediately as a local variable. Nevertheless, a choice is made here in the derivation. It depends on the problem at hand whether the derivation is started with a global or with a local postcondition. Often one starts with a specification for a sequential program. Therefore, it is natural to massage the postcondition in such a way that one easily obtains local invariants from it. This can be done by taking the data distribution into account.

We continue with the partial combine problem. Consider $P1.q\ (t := t+1)$, i.e., $P1$ with t replaced by $t+1$.

$$\begin{aligned} & P1.q\ (t := t+1) \\ \equiv & \quad \{ \text{ definition } P1.q\ (t := t+1) \ \} \\ & 0 \leq q < 2^{t+1} \Rightarrow m = M.0.(q+1) \\ \equiv & \quad \{ \text{ definition } P1, \text{ range splitting } \} \\ & P1.q \wedge (2^t \leq q < 2^{t+1} \Rightarrow m = M.0.(q+1))\,. \end{aligned}$$

$P1.q\ (t := t+1)$ equals $P1.q$ for $0 \leq q < 2^t$; only processes q in the range $2^t \leq q < 2^{t+1}$ have to compute $M.0.(q+1)$. For a process q, with $2^t \leq q < 2^{t+1}$, we have

$$\begin{aligned} & m = M.0.(q+1) \\ \equiv & \quad \{ \text{ definition } M \ \} \\ & m = (\odot\, i : 0 \leq i < q+1 : f(i)) \\ \equiv & \quad \{\ 2^t \leq q < 2^{t+1}, \text{ range splitting } \} \\ & m = (\odot\, i : 0 \leq i < q - 2^t + 1 : f(i)) \odot (\odot\, i : q - 2^t + 1 \leq i < q+1 : f(i)) \\ \equiv & \quad \{ \text{ definition } M \ \} \\ & m = M.0.(q - 2^t + 1) \odot M.(q - 2^t + 1).(q+1)\,. \end{aligned}$$

Here, we used the property

$$(*) \quad M.a.c = M.a.b \odot M.b.c,\ 0 \leq a < b < c \leq p\ ,$$

for the particular choice $a = 0$, $b = q{-}2^t{+}1$, and $c = q{+}1$. The value of $M.0.(q{-}2^t{+}1)$ is, on account of $P0$, known in process $(q - 2^t)$. The value of $M.(q - 2^t + 1).(q + 1)$ is, however, unknown. This suggests to strengthen P with invariant $P2$ in which the value of $M.(q - 2^t + 1).(q + 1)$ is recorded in m for all processes q with $q \geq 2^t$.

$$\begin{aligned} P.q &: P0.q \wedge P1.q \wedge P2.q \\ P2.q &: 2^t \leq q \Rightarrow m = M.(q - 2^t + 1).(q + 1)\,. \end{aligned}$$

Upon initialisation $m = M.(q{-}2^0{+}1).(q{+}1) = f(q)$ for $q > 0$, and $m = M.0.1 = f(0)$ for $q = 0$ needs to hold. Consider $P2.q\,(t := t + 1)$.

$$\begin{aligned} & P2.q\ (t := t + 1) \\ \equiv & \quad \{ \text{ definition } P2.q\ (t := t + 1) \ \} \\ & 2^{t+1} \leq q \Rightarrow m = M.(q - 2^{t+1} + 1).(q + 1) \\ \equiv & \quad \{ \text{ definition } M, \text{ range splitting } \} \\ & 2^{t+1} \leq q \Rightarrow m = M.(q - 2^{t+1} + 1).(q - 2^t + 1) \odot M.(q - 2^t + 1).(q + 1). \end{aligned}$$

Again we used property $(*)$ but with different a, b, and c. Term $M.(q - 2^t + 1).(q + 1)$ is known on account of $P2$. Term $M.(q - 2^{t+1} + 1).(q - 2^t + 1)$ is known in process $(q - 2^t)$. Note that for $P1$ the range $2^t \leq q < 2^{t+1}$ in $P2$ is sufficient, but when maintaining $P2$ one needs to enlarge the range to $2^t \leq q < p$. We used in this derivation a splitting rule expressed by property $(*)$.

It is now clear what needs to be done. There are three kind of processes: each process q with $0 \leq q < 2^t$ has trivially restored $P1$ and $P2$, each process q with $2^t \leq q < 2^{t+1}$ has to restore $P1$ with a proper value of process $(q - 2^t)$, and finally each process q with $2^{t+1} \leq q < 2^k$ has to restore $P2$ with a proper value of process $(q{-}2^t)$. Therefore, in iteration t, process q should engage in the receiving from process $(q - 2^t)$ and in the sending to process $(q + 2^t)$ if they exist. The resulting communication process is given in Figure 1.4.

In the annotated program $\hat{M}.q.t$ is shorthand for $M.(q - 2^t + 1).(q + 1)$ if $q \geq 2^t$, and for $M.0.(q + 1)$ otherwise.

The time complexity of the combine program is $\mathbf{O}(k) = \mathbf{O}(\log p)$, because in each iteration at most two communications and at most one $\odot$ operation take place in a process. The total number of communications is:

$$(\sum t : 0 \leq t < k : 2^k - 2^t) = (k - 1) * 2^k + 1\ ,$$

since there are $2^k - 2^t$ senders (and hence, receivers) in iteration t.

The program can be generalised to arbitrary p, not necessarily a power of two. The efficiency of the program is low,

$$E.p.n = \mathbf{O}(\tfrac{1}{\log p})\ ,$$

```
S.q ::
[[ m, t, x: int;
   t := 0 ; m := f(q)
  {P.q}
  ; do t ≠ k →
        if q < 2^t →  {m = M̂.q.t} (q + 2^t)!m
        [] 2^t ≤ q < 2^k − 2^t →
             {m = M̂.q.t}
              par (q − 2^t)?x , (q + 2^t)!m rap
             {x = M̂.(q − 2^t).t}
             ; m := m ⊙ x
        [] 2^k − 2^t ≤ q < 2^k →
              (q − 2^t)?x
             {x = M̂.(q − 2^t).t ∧ m = M̂.q.t}
             ; m := m ⊙ x
        fi
      ; t := t + 1 {P.q}
   od
]]
```

Figure 1.4: Outline of each process $S.q$ for the partial combine problem

and can be improved to $\mathbf{O}(1)$ using the techniques of [49].

The communication network used for the combine problem is a graph consisting of p nodes; node q is connected by an edge to nodes $q + 2^t$ for all t for which they exist. For $n = 16, k = 3$, the communication network is given in Figure 1.5. This network can be mapped on a binary hypercube as follows [48].

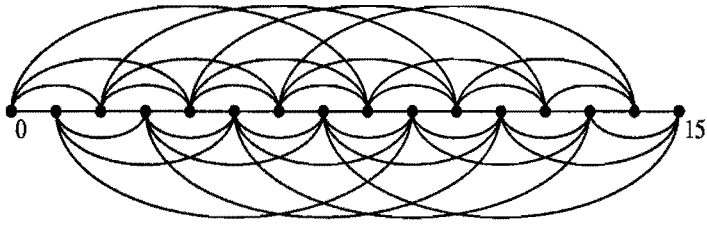


Figure 1.5: Communication network for partial combines using 16 processes

Let $[q]$ denote the binary representation of natural number q, $\#[q]$ is the number of 1's in $[q]$, and $*$ is taking bitwise exclusive-or, for example $[3] * [6] = 011 * 110 = 101$. Define the map g by $g.q = [q] * [q/2]$, then the following holds (without proof):

$$\#(g.q * g.(q + 2^t)) = \begin{cases} 1 & \text{if } t = 0 \\ 2 & \text{otherwise} \end{cases} \quad \text{for all } q, t\text{: } 0 \leq q + 2^t < 2^k .$$

This means that g maps two neighbouring nodes q and $q + 2^t$ in the communication network on two binary nodes in the hypercube which differ only in one position if $t = 0$ and in two positions if $t > 0$. Hence, in the terminology of [39] each edge of the communication network is mapped onto a path of length 1 or 2 in the hypercube. An example mapping is given in Figure 1.6.

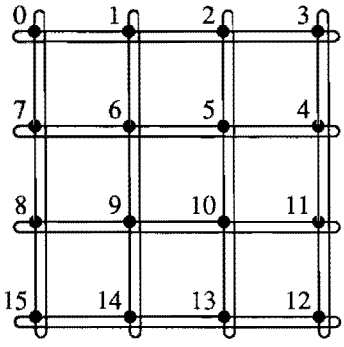


Figure 1.6: Mapping of the graph of Figure 1.5 onto a hypercube of 16 processes. The numbers are integer representation of the images $g.q$. Each edge in the original graph is mapped onto a path of length at most 2.

In conclusion, all partial combines can be effectively computed on a hypercube by a communication process with a time complexity of $\mathbf{O}(\log p)$.

1.6 The parseq rule

In this section we discuss a rule that lies at the heart of our design method; it stems from distributed protocol construction and verification. The observation made is that the logical structure of a protocol can often be described as a sequential composition of a number of parallel tasks each corresponding to a phase from the protocol. Examples are the PIF protocol described by [74], the distributed weighted-spanning-tree algorithm of Gallager, Humblet and Spira [22], and the shortest-path algorithm of [8].

This observation equally applies to parallel program construction and verification, and was as such first recognised by Elrad and Francez in [20]. They demonstrated that a parallel program can be decomposed into so-called *communication-closed* layers, i.e., communication between processes belonging to different layers does not occur. They considered a parallel program S (casting their definitions in our notation)

$$S :: \textbf{par } q : 0 \leq q < p : S.q \textbf{ rap}$$

in which every process $S.q$ can be represented as

$$S.q :: S_0.q \ ; \ \ldots \ ; \ S_{d-1}.q \ .$$

The $S_i.q$'s consist of simple statements: *skip*, assignment, and communication. Introduction of redundant *skip* statements allows for d, the depth of the decomposition, to be uniform over all processes $S.q$.

A layer of S, denoted by L_j, $0 \leq j < d$, consists of

$$L_j :: \textbf{ par } q : 0 \leq q < p : S_j.q \textbf{ rap } .$$

A layer is called communication closed iff communication actions taking place in that layer do not cross the boundary of the layer. Stated differently: any ! in a layer matches with a ? in the same layer and vice versa. The decomposition of S into layers is

$$S :: L_0 \ ; \ L_1 \ ; \ \ldots \ ; \ L_{d-1} \ .$$

The decomposition is called *safe* iff all layers are communication closed.

Elrad and Francez's main result is:

> *A distributed program is equivalent to any of its safe decompositions into layers.*

This is proven by induction on d, the depth of the decomposition, but unfortunately the step

$$L_{d-1} \ ; \ L_d \text{ "is equivalent to" } \textbf{ par } q : 0 \leq q < p : S_{d-1}.q \ ; \ S_d.q \textbf{ rap}$$

is unproven. For $p = 2$ this boils done to:

Definition 1.22 (parseq rule) Given are four terminating processes $S_i.j$, $0 \leq i, j < 2$, which have as *only* interactions:

> Each $S_i.0$ can interact via communication with $S_i.1$.
>
> Each $S_0.i$ can interact via shared variables with $S_1.i$.

it holds that

par $S_0.0$, $S_0.1$ **rap** ; **par** $S_1.0$, $S_1.1$ **rap**
"is equivalent to"
par $S_0.0$; $S_1.0$, $S_0.1$; $S_1.1$ **rap** .

□

This rule is called the **parseq** rule. The rule mentions the equivalence between two program fragments, which are called the left-hand composition and the right-hand composition. Two programs are considered equivalent if their pre- and postconditions are the same. In the **parseq** rule there are four processes that play a rôle; all four are composed in two different ways.

From the point of view of a single process $S_i.j$ both the left-hand side composition and the right-hand composition in the **parseq** rule define the same sequence of computations in $S_i.j$. A similar remark holds for the processes $S_0.0$; $S_1.0$ and $S_0.1$; $S_1.1$. Hence "is equivalent to" means that each program $S_i.j$ and $S_0.j$; $S_1.j$ when started in a state described by the precondition, reaches a state described by the postcondition. The sequence of possible computations is the same, but the order in which they happen may differ. For example, in the right-hand composition process $S_1.0$ can already start after termination of process $S_0.0$. In the left-hand composition process $S_1.0$ can only start after termination of both $S_0.0$ and $S_0.1$. This behaviour can be observed only by an external observer; it cannot be observed by process $S_1.0$ itself, since there is no interaction via communications between process $S_0.0$ and $S_0.1$. Operationally speaking, the **parseq** rule states that global synchronisation can be removed at the expense of strict interaction rules. The order of computations in the processes can be expressed formally in trace theory [69]. Trace theory also allows the **parseq** rule to be restated and proved, as has been shown by J.J. Lukkien. Other proofs of the **parseq** rule are based on temporal logic [74].

Our interest in this rule is that we believe to reflect the way we construct parallel programs. Our parallel programs consist of p instances of a single parameterised process. Such a process is designed by refining it into a sequence of sequential programs or communication processes. All instances of a parameterised process in this sequence form a layer, and the layers are syntactically separated by semicolons. The parameterised processes belonging to each layer are specified by preconditions and postconditions, and can be studied in isolation. This strict design principle still allows us to design efficient parallel programs, since the **parseq** rule shows a way out from a strict synchronisation between layers.

Example 1.23 (PIF) In the PIF protocol there is a tree of processes, and the root (process) informs all other processes about a message m. The root has to be informed that all other processes have received m. We can identify two phases in the program. In the first phase, a global broadcast of message m from the root to all the other processes takes place, in the same way as in the communication process for the **sum** problem (see Figure 1.3). In the second phase, a communication process is started which informs the root about global receipt of m. By designing two parameterised communication processes for each phase we can implement the PIF protocol in a sequential programming style. We thus obtain a layered parallel program. In the execution of the parallel program it might well happen that a process starts with the second phase while other processes are still in the first phase. Nevertheless, each process will interact only with processes executing in the same phase. □

1.7 Summary

We described in this chapter the main ingredients of parallel program design. Our method can be roughly summarised as follows.

- Functional specification.
  Aim: Formulation of a parameterised functional specification from an ordinary sequential specification. Introduction of p processes, each process is assigned a part of the work involved.

  Concerns: How to split up the specification into p local ones. Is a data distribution given, or part of the problem? Can we postpone the choice of distribution?

- Invariants.
  Aim: Obtain from a local specification of a parameterised process a parameterised invariant. Introduce subproblems by using the *layer* concept. Give parameterised specifications of each layer. Design sequential programs for the non-communicating parts of a layer using standard techniques.

  Concerns: Can we discriminate easily the functionality of communication processes? Avoidance of communication processes.

- Communication processes.
  Aim: Obtain a local specification of a communication process. Formulate the communication requirements in a communication-network independent way.

  Concerns: How can we realise the communication processes on different communication networks? What is the impact of the communication networks on the total time complexity?

- Distribution of data or work.
  Aim: Minimise the number of steps to complete the parallel program. For example, balance the load as best as possible across the p processes.

  Concerns: Are alternative distributions possible? Can we determine the communication complexity, and the influence of the distribution on the communication requirements? Is it possible to avoid structural load imbalances?

- Complexity
  Aim: To determine a complexity formula in the size of the input and the number of processes used.

  Concerns: Are we able to give the complexity of alternative solutions? Are the final algorithms scalable, i.e., can high speed-up results be obtained and is there experimental evidence?

In the remainder of this monograph we will follow this method. It is our primary concern to obtain a correct parallel program by using a decomposition of the program into layers of parameterised processes. By studying possible data distributions for the problem involved, we can analyse the work load distribution with the aim of obtaining efficient programs. Data distributions are the topic of the next chapter.

Chapter 2

Distributions

In this chapter, we consider simple distributions of arrays and matrices. The purpose of this study is to focus on the work-load properties of these distributions. Furthermore, the impact of distributions on the number of communications is discussed.

2.0 Introduction

The parallel programs we consider use distributed data objects, like arrays or graphs. Each part of a data object is assigned to a unique process. In a parallel program, processes can independently perform operations on their local data until global information is needed such as, for instance, the value of a global sum. The communication processes are responsible for combining and collecting global information via message passing. Processes send messages along channels, and a message can only be delivered when the receiving process is ready. This causes synchronisation points in the parallel program, which usually lead to waiting times. As a consequence, the parallel program will have a lower efficiency.

One of the targets of parallel programming is to obtain programs with a high speed-up. In other words: the more processes there are, each executing on a different processor, the faster the parallel program should run for a certain input. The impact of distributions on the time complexity of a parallel program is therefore an important issue.

In this chapter we primarily want to focus on simple static distributions and their properties. The general mapping problem is not considered.

A static distribution assigns parts of a data object (an element) to a process; the assignment does not change during the execution of the program. Important properties of static distributions are: the number of elements that are assigned to a process and the distribution of elements across the processes. On the basis of these properties, we can compare distributions and quantify the influence of distributions on the work load in a parallel program.

(Representation details of distributed data will not be addressed. An efficient representation depends on the operations to be performed; this is not the issue here.)

From simple distributions we can make new distributions by composition and Cartesian product (see Sections 2.2 and 2.3). It turns out that both kinds of distribution have similar properties.

Distributions also determine which values have to be communicated in a process. It is desirable to count the number of communications for a given distribution. For that purpose a counting technique is presented in Section 2.4. The advantage of this technique is that alternative distributions can be judged on their communication overhead. In an actual parallel program, we strive for spreading the communications across the processes, since in this way we can reduce the communication complexity.

To end this section we want to stress the double rôle of distributions in parallel programs. The main rôle is to distribute the work across the processes in such a way that the completion time of the parallel program is minimised. This can be done, for example, by avoiding structural load *imbalances*. The other rôle of distributions is their influence on the number of communications in a parallel program. This double rôle can be more easily identified when there is a clear distinction between communication processes and non-communicating programs, as is the case in all our parallel programs.

2.1 One-dimensional distributions

This section discusses a number of frequently used distributions of arrays together with their properties. In the following, the notation $\mathbf{K}$ is used to denote the set $\{0 \ldots K-1\}$, $K \geq 0$.

Definition 2.0 (distribution) $\mathcal{D} = (\delta, A, B)$ is called a distribution if A and B are finite sets, and δ is a mapping from A to B. □

The domain A and range B of δ are mentioned explicitly. Set A specifies the set of data objects of interest; set B specifies the set of processes, which is usually $\mathbf{p}$. Since the distributions are static, it is possible that there are more processes than data objects to perform work on. Such a situation often occurs at the end of a computation, when some processes have already terminated.

The distribution of an array f of length n across p processes can be specified by the triple $(\delta, \mathbf{n}, \mathbf{p})$. The data object of interest is an array of which the array elements are identified by their index set $\mathbf{n}$. The function δ assigns each array index $i \in \mathbf{n}$ (and its corresponding array element $f(i)$) to a process number from set $\mathbf{p}$.

Well-known ways of distributing an array are: every element to one unique process (identity), assigning p equally-sized consecutive array segments (linear, consecutive storage) and assigning elements cyclically (wrap, cyclic storage) [44].

Example 2.1 (*identity, linear, wrap*)

$$\begin{aligned} \mathit{identity} &= (\,(\lambda i \cdot i), \mathbf{p}, \mathbf{p}\,)\,, \\ \mathit{linear} &= (\,(\lambda i \cdot i/(n/p)\,), \mathbf{n}, \mathbf{p}\,)\,, \text{ provided that } p|n\,, \\ \mathit{wrap} &= (\,(\lambda i \cdot i \backslash p), \mathbf{n}, \mathbf{p}\,)\,. \end{aligned}$$

The lambda notation is used in the definition of a distribution function. □

We have the freedom to permute the process numbers. This is not essential for our purposes, since properties of distributions, like the maximal number of data objects assigned to a process, are invariant under such a permutation.

Definition 2.2 (equality of distributions)

$$(\delta 0, \mathbf{n}, \mathbf{p}) = (\delta 1, \mathbf{n}, \mathbf{p}) \equiv (\exists\ \pi : \pi \text{ a permutation on } \mathbf{p} : \delta 0 = \pi \circ \delta 1)\,,$$

where $\circ$ denotes composition. □

In the following example, it is shown that a permutation on the set of data objects can cause two distributions to become equal.

Example 2.3 Consider the *linear* and *wrap* distribution of an array, and define the permutation σ on $\mathbf{n}$ by:

$$\sigma = (\lambda i \cdot (i \backslash m) * p + i/m), \text{ where } m = n/p\,,$$

then

$$\delta^{linear} = \delta^{wrap} \circ \sigma$$

holds. □

Permuting the set of data objects before applying a distribution function effectively means picking another distribution.

In the definition of *linear* distribution of Example 2.1, the restriction $p|n$ was imposed. A general *linear* function can be obtained as follows.

Example 2.4 (*linear*) For a *linear* distribution, the remaining $n \backslash p$ indices of an array of length n can be distributed by assigning the first $n \backslash p$ processes one array element extra. Formally, the distribution function becomes:

$$\begin{aligned} \mathit{linear} &= (\delta, \mathbf{n}, \mathbf{p}), \\ \delta &= (\lambda i \cdot i/(m+1)\ \mathbf{max}\ (i - n \backslash p)/m\ ), \text{ and } m = n/p\,. \end{aligned}$$

Note that if $p|n$ then the distribution function for *linear* is recovered, since $i/m \geq i/(m+1)$ for all $i \geq 0$. The formula for this general *linear* distribution is very compact. □

In the following definition, distributions are used in determining the sets of local variables assigned to a process.

Definition 2.5 (owns) Given is a distribution $(\delta, \mathbf{n}, \mathbf{p})$. The set of elements $\mathbf{n}$ assigned to process q, $0 \leq q < p$, is given by $\mathcal{O}.q$ (pronounced "owns"):

$$\mathcal{O}.q = \{i : i \in \mathbf{n} \wedge \delta.i = q : i\} \ .$$

□

The notation $\mathcal{O}^{\delta}.q$ refers to the set $\mathcal{O}.q$ with a specific function δ in mind. Usually, the sets $\mathcal{O}.q$ are used to define a distribution. The sets $\mathcal{O}.q$ form a partition of $\mathbf{n}$.

Example 2.6 (ℓ) Another characterisation of the general *linear* function is as follows:

$$\mathcal{O}^{linear}.q = \{i : \ell.q \leq i < \ell.(q+1) : i\} \ ,$$

where

$$\ell = (\lambda q \cdot q * (n/p) + q \ \mathbf{min} \ (n \backslash p)) \ .$$

□

Each process has a number of array elements assigned to it. Each array element, or part of a data object in general, has an associated number of operations to perform on it. Counting the cardinalities of the sets $\mathcal{O}.q$ is therefore meaningful in time-complexity analysis. In this way, a good indication of the amount of work per process is obtained. Often, we are interested in the maximal number of data objects that is assigned to a process, because it usually determines the time complexity of a parallel program. (This assumption holds only if the amount of work per data object is constant.)

Definition 2.7 (Ma) The maximum number of data objects assigned to a process for a distribution $(\delta, \mathbf{n}, \mathbf{p})$ is defined by

$$Ma(\delta) = (\mathbf{max}\ q : 0 \leq q < p : \mid \mathcal{O}^{\delta}.q \mid) \ .$$

□

Example 2.8 ($Ma(linear) = Ma(wrap)$) For the array distributions *linear* and *wrap* the following holds:

$$\mid \mathcal{O}^{linear}.q \mid = \mid \mathcal{O}^{wrap}.q \mid = (n + p - 1 - q)/p \ .$$

The cardinalities for *linear* can be obtained using the ℓ function (Example 2.6). The cardinalities for the *wrap* distribution are obtained from:

$$\begin{array}{ll} & \mid \mathcal{O}^{wrap}.q \mid \\ = & \quad \{ \text{ definition } \} \\ & \mid \{i : 0 \leq i < n \wedge i \backslash p = q : i\} \mid \\ = & \quad \{ \text{ calculus, range splitting } \} \\ & \mid \{i : 0 \leq i < (n/p) * p \wedge i \backslash p = q : i\} \mid + \mid \{i : (n/p) * p \leq i < n \wedge i \backslash p = q : i\} \mid \\ = & \quad \{ \text{ calculus } \} \\ & n/p + \mid \{i : 0 \leq i < n \backslash p \wedge i = q : i\} \mid \\ = & \quad \{ \text{ calculus } \} \\ & (n + p - 1 - q)/p \ . \end{array}$$

The cardinalities of the $\mathcal{O}.q$'s are the same for the *linear* and *wrap* distributions. Both distributions assign a maximal number of elements to process 0:

$$Ma(linear) = Ma(wrap) = (n + p - 1)/p \ .$$

Hence, if the time complexity of a parallel program is determined entirely by the cardinalities of one of these distributions, then there is no difference between *linear* and *wrap*. □

A measure of the load *imbalance* is the difference between the maximum number of elements assigned to a process and the minimum number of elements assigned to a process. If this measure is bounded by a natural number w for a given distribution then we call this distribution w-balanced. Of course, we are only interested in small values of w.

Definition 2.9 (w-balanced) A distribution $(\delta, \mathbf{n}, \mathbf{p})$ is called w-balanced, $w \geq 0$, iff

$$Ma(\delta) - Mi(\delta) \leq w,$$

where

$$\begin{aligned} Ma(\delta) &= (\mathbf{max}\ q : 0 \leq q < p : |\mathcal{O}^{\delta}.q|) \\ Mi(\delta) &= (\mathbf{min}\ q : 0 \leq q < p : |\mathcal{O}^{\delta}.q|) \ . \end{aligned}$$

□

A w-balanced distribution ensures that the differences in the work-load distribution between processes are bounded by w. If this number is small then we have ensured a good load balance. A distribution is called *homogeneous* if $w = 1$ [39] and *perfect* if $w = 0$. A perfect distribution is also homogeneous. It is clear that a *perfect* distribution is only achieved when $p|n$, and this is in general not the case. Examples of *homogeneous* distributions are the already-mentioned *linear* and *wrap* distributions.

Example 2.10 (reflection) An example of a non-homogeneous distribution is

$$(\delta, \mathbf{n}, \mathbf{p}),\ \text{with}\ \delta = (\lambda i \cdot \begin{cases} i \backslash p & \text{if } (i/p) \backslash 2 = 0 \\ p - 1 - i \backslash p & \text{otherwise} \end{cases}) \ .$$

This distribution is called reflection. It is w-balanced with $w = 2$. □

For a certain class of computations the *wrap* distribution is a good candidate. To demonstrate this consider the following example.

Example 2.11 Given is a parallel program consisting of n steps. In step k, $0 \leq k < n$, computations are done only for the first k elements of the program's arrays (all arrays have length n and are distributed in the same way). Each array element requires a constant number of elementary operations. The maximal number of computations in step k in a process for any array distribution $(\delta, \mathbf{n}, \mathbf{p})$ is bounded from below by:

$$(\mathbf{max}\ q : 0 \leq q < p : |\mathcal{O}^{\delta}.q \cap \mathbf{k}|) \geq (k + p - 1)/p \ .$$

This follows from:

$$
\begin{array}{ll}
& p * (\mathbf{max}\ q : 0 \leq q < p : |\mathcal{O}.q \cap \mathbf{k}|) \\
\geq & \quad \{ \text{ calculus } \} \\
& (\sum q : 0 \leq q < p : |\mathcal{O}.q \cap \mathbf{k}|) \\
= & \quad \{ \mathcal{O}.q \text{ forms a partition of } \mathbf{k} \} \\
& |(\cup\ q : 0 \leq q < p : \mathcal{O}.q \cap \mathbf{k})| \\
= & \quad \{ \text{ calculus } \} \\
& k\ .
\end{array}
$$

The lower bound is attained by the *wrap* distribution (see Example 2.8, with n replaced by k). Hence, combining the results gives, for all δ:

$$(\mathbf{max}\ q : 0 \leq q < p : |\mathcal{O}^{\delta}.q \cap \mathbf{k}|) \geq (\mathbf{max}\ q : 0 \leq q < p : |\mathcal{O}^{wrap}.q \cap \mathbf{k}|)\ .$$

Or in words: at any step in such a parallel program, the maximum number of computations in a process for any distribution is at least the maximum number of computations in a process when using the *wrap* distribution.

If all processes synchronise at every step then the maximum number of computations performed by a process determines the computation time of a step. The work load is *homogeneously* distributed in every step for a *wrap* distribution. Therefore, we can conclude that for this class of computations the *wrap* distribution is a good candidate (which does not exclude the existence of other distributions with similar properties).

If the computations can overlap in different steps, i.e., a process can start step $k+1$ immediately after finishing step k, then the total load *imbalance* when using the *wrap* distribution is $\mathbf{O}(n)$. This result can be obtained by comparing the minimum and maximum over all q of the expressions:

$$
\begin{array}{ll}
& (\sum k : 0 \leq k < n : |\ \mathcal{O}^{wrap}.q \cap \mathbf{k}\ |) \\
= & \quad \{ \text{ Example 2.8 } (n := k) \} \\
& (\sum k : 0 \leq k < n : (k + p - 1 - q)/p) \\
= & \quad \{ \text{ calculus } \} \\
& (n * (n-1))/(2 * p) + \mathbf{O}((n * (p - 1 - q))/p\ .)
\end{array}
$$

The total amount of work is $\mathbf{O}(n^2)$ and (ideally) per process $\mathbf{O}(\frac{n^2}{p})$ (assuming that $p \leq n$). This means that for overlapping computations the *wrap* distribution has good characteristics, since the load imbalance is at most the amount of work per process.

Note that a similar argument holds for parallel programs that perform computations on the last $n - k$ array elements. Or, even more general, for programs that perform computations for different consecutive parts of arrays. □

2.2 Composition of distributions

Distribution functions can be composed to obtain new distributions.

Example 2.12 A distribution called *wrap-of-linear* can be obtained as follows. Take $linear = (\delta^{linear}, \mathbf{n}, \mathbf{m})$, $m|n$, and $wrap = (\delta^{wrap}, \mathbf{m}, \mathbf{p})$ then

$$wrap \circ linear = (\ (\lambda i \cdot (i/(n/m))\backslash p), \mathbf{n}, \mathbf{p})\ .$$

For $m = n$ the *wrap* distribution is obtained and for $m = p$ the *linear* distribution. For simplicity, we require $m|n$; a similar formula can be given for the generalised *linear* distribution. □

Definition 2.13 (composition of distributions) The composition of two distributions $\mathcal{D}0 = (\delta 0, \mathbf{m}, \mathbf{M})$, $\mathcal{D}1 = (\delta 1, \mathbf{n}, \mathbf{N})$, with $M = n$ is defined by:

$$\mathcal{D}1 \circ \mathcal{D}0 = (\delta 1 \circ \delta 0, \mathbf{m}, \mathbf{N})\ .$$

□

Other examples of composition are the variations *linear-of-wrap*, or *linear-of-linear*. Composition allows for making complex distributions. The properties of a composed distribution can be obtained from the properties of its constituents.

Example 2.14 Consider $wrap \circ linear$ of Example 2.12. The number of elements assigned to a process q, $\mid \mathcal{O}^{wrap \circ linear}.q \mid$, can be obtained by counting:

$$\mid \{i, j : 0 \leq i < m \wedge \delta^{wrap}.i = q \wedge 0 \leq j < n \wedge \delta^{linear}.j = i : j\} \mid\ .$$

This expression can be rewritten to

$$(\sum i : 0 \leq i < m \wedge i\backslash p = q : \ell.(i+1) - \ell.i)\ ,$$

which is for $m \mid n$ equal to $\frac{m}{n} * (m + p - 1 - q)/p$. This distribution is $\frac{m}{n}$ balanced if $m\backslash p > 0$, otherwise it is perfect. □

Composition of distributions does not always preserve homogeneity.

Example 2.15 Take the composition of two generalised *linear* functions, with $n = 12, M = m = 5, N = 3$, thus $\delta 0 : \{0 \ldots 11\} \rightarrow \{0 \ldots 4\}$ and $\delta 1 : \{0 \ldots 4\} \rightarrow \{0 \ldots 3\}$. Then the result of $\mathcal{D}0$ is:

$$\underbrace{0\,1\,2}\ \underbrace{3\,4\,5}\ \underbrace{6\,7}\ \underbrace{8\,9}\ \underbrace{10\,11}\ ,$$

and $\mathcal{D}1 \circ \mathcal{D}0$ results in:

$$\underbrace{0\,1\,2\,3\,4\,5}\ \underbrace{6\,7\,8\,9}\ \underbrace{10\,11}\ .$$

Here $\underbrace{9\,10}$ denotes the process that contains elements 9 and 10. □

What can be said of $|\mathcal{O}^{\delta 1 \circ \delta 0}.q|$ for two homogeneous distributions $\mathcal{D}0 = (\delta 0, \mathbf{m}, \mathbf{M})$, $\mathcal{D}1 = (\delta 1, \mathbf{n}, \mathbf{N})$, with $M = n$? Clearly the maximum number of elements assigned is at most $Ma(\delta 0) * Ma(\delta 1)$:

$$Ma(\delta 1 \circ \delta 0) = Ma(\delta 0) * Ma(\delta 1) = (m + M - 1)/M * (n + N - 1)/N,$$

and the minimum number assigned is at least $Mi(\delta 0) * Mi(\delta 1)$:

$$Mi(\delta 1 \circ \delta 0) = Mi(\delta 0) * Mi(\delta 1) = m/M * n/N \ .$$

Thus we arrive at the following result.

For homogeneous distributions $\mathcal{D}0 = (\delta 0, \mathbf{m}, \mathbf{M})$, $\mathcal{D}1 = (\delta 1, \mathbf{n}, \mathbf{N})$, $M = n$:

$$Mi(\delta 1) * Mi(\delta 0) \leq |\mathcal{O}^{\delta 1 \circ \delta 0}.q| \leq Ma(\delta 1) * Ma(\delta 0) \ .$$

The distribution $\mathcal{D}1 \circ \mathcal{D}0$ is w-balanced with

$$w = 1.n.N * (m/M) + 1.m.M * (n/N) + (1.m.M) * (1.n.N) \ ,$$

where the function $1.a.b$ yields 1 if $a \backslash b > 0$ and 0 otherwise.

Practical applications of composed distributions are, for example, parallel programs using different data distributions. By introducing a parameter like m in *wrap-of-linear*, it is possible to trade off the load balance for each individual part and to avoid expensive redistributions during a computation.

On the other hand, these distributions often yield complex expressions and are therefore not very well suited in program derivations. In the next section, we encounter another way of making new distributions.

2.3 Cartesian distributions

Distributions of multi-dimensional arrays can be modeled by Cartesian distributions. We will consider arrays of arrays, or matrices. In the following, it assumed that an m by n matrix is distributed across p processes.

Definition 2.16 (Cartesian distribution) The Cartesian product of two array distributions $\mathcal{D}0 = (\delta 0, \mathbf{m}, \mathbf{M})$, $\mathcal{D}1 = (\delta 1, \mathbf{n}, \mathbf{N})$, is defined by:

$$\mathcal{D}0 \times \mathcal{D}1 = (\delta 0 \times \delta 1, \mathbf{m} \times \mathbf{n}, \mathbf{M} \times \mathbf{N}) \ .$$

Where the function $\delta 0 \times \delta 1$ assigns to every index pair a pair of process numbers. Formally,

$$\delta 0 \times \delta 1 = (\lambda i, j \cdot (\delta 0.i, \delta 1.j)) \ .$$

□

The Cartesian product of two one-dimensional distributions uses a process pair as identification for a process. In order to obtain a process number an additional mapping $\beta : M \times N \rightarrow p$, $p = M * N$, must be applied. The function β is a bijection, which identifies a pair of process numbers with a process number in $\mathbf{p}$. We often omit a suitable β and use an integer pair as process identification in programs with matrix distributions.

Cartesian distributions of matrices can be obtained by distributing the rows of the matrix independently from the columns. The most commonly used matrix distributions are Cartesian. The set of elements assigned to each process by a Cartesian distribution $\mathcal{D}0 \times \mathcal{D}1$ of an m by n matrix can be defined in a similar as in Definition 2.5. It has the following property:

$$\mathcal{O}^{\delta 0 \times \delta 1}.(s,t) = \mathcal{O}^{\delta 0}.s \times \mathcal{O}^{\delta 1}.t \ ,$$

with $0 \leq s < M$ and $0 \leq t < N$.

In our opinion, non-Cartesian distributions are often more difficult to program due to the absence of a "splitting-rule" like the one above. Cartesian distributions give us the freedom to consider the rows and columns as entire identities. Since p is fixed, we can consider all decompositions of M and N such that $p = M * N$ holds. This gives an additional degree of freedom in the derivations.

Consider two *homogeneous* distributions $\mathcal{D}0$ and $\mathcal{D}1$. Similar results hold for the Cartesian distribution $\mathcal{D}0 \times \mathcal{D}1$ as for composition, for instance,

$$Mi(\delta 0) * Mi(\delta 1) \leq |\mathcal{O}^{\delta 0 \times \delta 1}.(s,t)| \leq Ma(\delta 0) * Ma(\delta 1) \ ,$$

with Mi and Ma as defined in the previous section.

Cartesian distributions are easier to handle in program derivations than composite distributions. Their usage is discussed in Chapter 4. In the following example we give some frequently-used Cartesian distributions.

Example 2.17 (Cartesian distributions) Let $p = M * N$ and consider the following Cartesian distributions:

$$\begin{array}{ll}
linear^2 & = (\delta^{linear}, \mathbf{m}, \mathbf{M}) \times (\delta^{linear}, \mathbf{n}, \mathbf{N}) \text{ with } M = N. \\
row & = linear^2 \text{ with } N = 1. \\
col & = linear^2 \text{ with } M = 1. \\
wrap^2 & = (\delta^{wrap}, \mathbf{m}, \mathbf{M}) \times (\delta^{wrap}, \mathbf{n}, \mathbf{N}) \text{ with } M = N. \\
wrap\text{-}row & = wrap^2 \text{ with } N = 1. \\
wrap\text{-}col & = wrap^2 \text{ with } M = 1.
\end{array}$$

These six distributions are visualised in Figure 2.0. The $wrap^2$ is introduced in [75] as the *grid* distribution. The $linear^2$ with $M = N$ is called *block* distribution.

□

2.4 Counting communications

Another aspect of distributions is their impact on the number of communications. During a computation processes need values which are not available locally, i.e., values which have been assigned to different processes, and hence have to be communicated. The distribution determines the total number of communications. Minimisation of

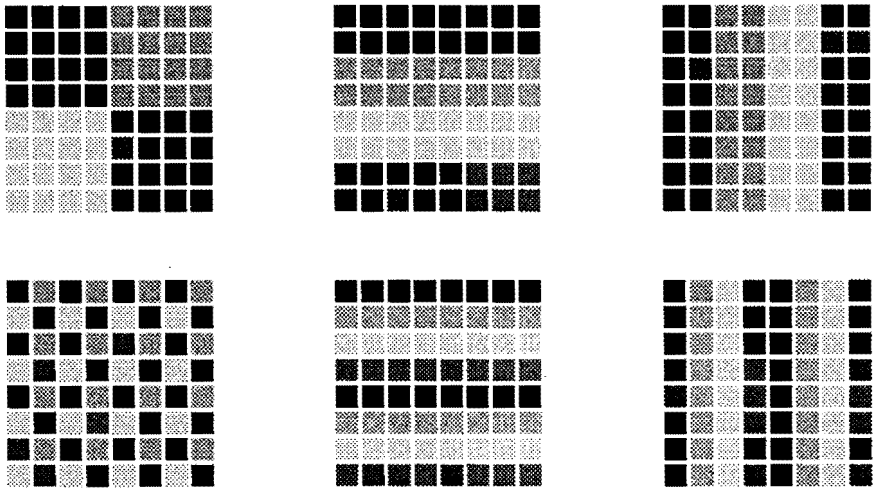


Figure 2.0: Six distributions: $linear^2$ = *block*, *row*, *col* (top), $wrap^2$ = *grid*, *wrap-row* and *wrap-col* (bottom). The grey-shading of a matrix element denotes the process to which the element has been assigned ($p = 4$, $m = n = 8$).

this number may lead to a low communication overhead. It is also important that the communications are spread evenly across the processes in such a way that many communications take place in parallel. The latter can only be achieved if the communication network offers enough freedom to implement the communication processes efficiently.

Given a program's postcondition and a distribution we can count the *total* number of communications. The program's postcondition is split in p local postconditions according to the distribution used. With every local postcondition a process is associated that will establish it. If it is assumed that every datum is assigned to one unique process then the total number of postconditions that refer to a particular datum is a measure of the number of communications of that datum. However, it may happen that a subexpression containing several data occurs in different postconditions. One process can compute such a subexpression and store the result in a variable, which is communicated to the other processes. In this way, communication is reduced. In order to have a meaningful interpretation of the counting technique we present here, we exclude the previously mentioned case.

We introduce for every datum e the quantity $Nocc.e$,

$$Nocc.e = \text{the number of local postconditions in which } e \text{ occurs} .$$

Since every e is assigned to a process it needs only to be communicated to $Nocc.e - 1$ other processes. By summing over all e we obtain the total number of communications $Ncom$,

$$Ncom = (\sum\ e :: Nocc.e - 1\ ) .$$

The value of *Ncom* is only determined by the way the program's postcondition is split up and the distribution used. The communication complexity $T_c.p.n$ is bounded from below by $(Ncom + p - 1)/p$ if a process can perform only one communication action at each moment.

This technique of counting communications allows us to compare distributions on the basis of their communication overhead. It is not always possible to count *Ncom* from a postcondition due to common subexpressions. For example, in computing all partial sums of a given array there are several subexpressions, namely, the partial sum of the first i elements is part of the partial sum of the first j, for $0 \leq i < j$. The applicability of the technique clearly depends on the problem at hand. Nevertheless, the results that can be obtained are independent of *any* communication network. To illustrate this technique we give an example.

Example 2.18 Given two matrices a and b, of dimensions $m \times o$, and $o \times n$, respectively. The problem is to compute matrix c, $m \times n$, satisfying postcondition R,

$$R \ : \ c = a\,b \ .$$

We use a Cartesian distribution $\mathcal{D}0 \times \mathcal{D}1$ for the matrix c, and introduce $p = M * N$ processes; each process is identified by an ordered pair (s,t), $0 \leq s < M$, $0 \leq t < N$. The local postcondition $R.s.t$ of process (s,t) becomes:

$$\begin{array}{rl} R.s.t \ : \ (\forall\, i,j \ : & 0 \leq i < m \wedge 0 \leq j < n \wedge \delta 0.i = s \wedge \delta 1.j = t \\ : & c(i,j) = (\sum\, k : 0 \leq k < o : a(i,k) * b(k,j)\ ))\ . \end{array}$$

Note that

$$(\forall\, s,t : 0 \leq s < M \wedge 0 \leq t < N : R.s.t) \Rightarrow R\ .$$

In order to count the number of communications, we introduce quantities $Nocc.a(i,k)$ and $Nocc.b(k,j)$.

$$\begin{array}{ll} & Nocc.a(i,k) \\ = & \\ & |\ \{s,t : 0 \leq s < M \wedge 0 \leq t < N \wedge \delta 0.i = s \wedge (\exists\, j :: \delta 1.j = t) : (s,t)\}\ |\ . \end{array}$$

Additionally, we require surjectivity of $\delta 1$.

Hence,

$$\begin{array}{ll} & Nocc.a(i,k) \\ = & \quad \{\ \text{definition } Nocc,\ \delta 1 \text{ surjective }\ \} \\ & |\ \{s,t : 0 \leq s < M \wedge 0 \leq t < N \wedge \delta 0.i = s \wedge true : (s,t)\}\ | \\ = & \quad \{\ \text{calculus }\ \} \\ & N* |\ \{s : 0 \leq s < M \wedge \delta 0.i = s : s\}\ | \\ = & \quad \{\ \delta 0 \text{ is a function }\ \} \\ & N\ . \end{array}$$

And in a similar way we obtain:

$$Nocc.b(k,j) = M \ .$$

if $\delta 0$ is surjective.

Summing over all $a(i,k)$ and $b(k,j)$ gives the total number of communications *Ncom*:

$$\begin{array}{ll} & Ncom \\ = & \quad \{ \text{ definition } Ncom \ \} \\ & (\sum i,k : 0 \leq i < m \wedge 0 \leq k < o : Nocc.a(i,k) - 1 \ ) + \\ & (\sum k,j : 0 \leq k < o \wedge 0 \leq j < n : Nocc.b(k,j) - 1 \ ) \\ = & \quad \{ \text{ calculus } \} \\ & o * (m * (N-1) + n * (M-1)) \ . \end{array}$$

Hence, the total number of communications *Ncom* for a matrix multiplication is:

$$Ncom = o * (m * (N-1) + n * (M-1)) \ ,$$

using any surjective Cartesian distribution $\mathcal{D}0 \times \mathcal{D}1$ of the $m \times n$ matrix c.

A number of observations can be made. For $M = N = p = 1$, $Ncom = 0$ and no communications are necessary. Furthermore, *Ncom* is independent of particular choices for $\delta 0$ and $\delta 1$. Since for any problem m, n, o and p are fixed we can determine M and N, $p = M * N$, such that *Ncom* is minimal. There are at least two such pairs namely $(1,p)$ and $(p,1)$. Clearly all possible values (M,N) are integer points on the hyperbola $p = M * N$, $1 \leq M,N \leq p$, and the values of *Ncom* for fixed m, n, and o, lie on the line with a slope dependent on $\frac{m}{n}$. Hence, the minimal value for *Ncom* depends on the ratio $\frac{m}{n}$ and in particular for $m = n$, *Ncom* has a minimal value if p is a square. The latter has also been observed in [21]. □

2.5 Final remarks

In this chapter, we discussed static distributions of arrays and matrices. New complex distributions can be obtained by composition and Cartesian product. The properties of these newly-formed distributions often allow more freedom in exploiting the properties of their constituents. For example, Cartesian distributions have the property to consider rows and columns as entire units distributed across M ensembles of N processes, respectively. Additionally, M and N may vary under the constraint $p = M * N$. On the other hand, composite distributions usually trade off its constituent properties, as is the case in the *wrap* ∘ *linear* distribution (Example 2.12).

Quantities like the sizes of the set elements assigned to a process allow for a comparison of the load-balancing properties of distributions. These counting techniques are very powerful in complexity analysis of parallel programs. A demonstration of these techniques has been given in Example 2.11, where it has been shown that the

wrap distribution is to be preferred over any other distribution for a class of parallel programs. The load-balancing properties of Cartesian distributions are easily derived from their constituents. In this context, Cartesian distributions are easier to deal with than composite distributions. Important measures of load balance are the maximum and minimal number of elements assigned to a process, and the difference between these two numbers (w-balancedness).

Distributions also have a certain impact on the total number of communications. Counting communications (another counting technique), enables evaluation of different distributions. The results obtained are independent of any communication network, and applying this technique usually results in a lower bound on the communication complexity. As discussed in Section 2.4 some requirements have to be met. Counting the total number of communications in a parallel program gives insight but it is not the only aspect. For instance, the spread of communications across the processes also has to be taken into account.

Another aspect which has not been addressed is the problem of distributed data representation. Since data is distributed across processes, each process has to represent its local parts in a data structure. An efficient representation depends, of course, on the operations to be performed. For the local parts of arrays and matrices simple parameterised data structures can be obtained in the form of local arrays and matrices. In general, attention to representations should be given only at the implementation level.

The rôle of distributions is an important one in parallel programming. Although our primary concern is to obtain a correctly behaving parameterised parallel program, we cannot ignore possible choices for the distribution of data. In practice, we often obtain hints about candidate distributions in a parallel program derivation. Examples of program derivations are given in the next two chapters.

Chapter 3

Parallel Segment Computations

A large example of the use of parameterised invariants is given. The target is to obtain a parallel program for a class of segment problems. Suitable distributions for the program are found during the derivations.

3.0 Introduction

We quote from [45]:

> *Segment problems were originally invented at the Eindhoven University to serve as exercises and exams in programming courses.*

A segment problem usually refers to a computation of a function defined on consecutive parts of an array. Well-known segment problems are the longest plateau and the maximal segment sum [33, 34].

In this chapter, a class of segment problems is considered that can be efficiently solved in parallel. One could argue that segment problems are artificial, and that there is little practical value in solving these problems in parallel. The derivation of parallel segment problems, however, demonstrates nicely the kind of difficulties encountered in parallel program design. Rather than just solving one or two segment problems, we outline a general parallel program scheme for the problem class.

An essential step in the design is the explicit formulation of divide-and-conquer rules that express the relation between the computation on segment $[a, c)$ and the computation on segments $[a, b)$ and $[b, c)$, where b is any interior point. It turns out that the application of the rules to the segments and the parallel program scheme itself are conveniently expressed by the same operator $\odot$.

The resulting parallel program scheme consists of a computation and a communication phase. This distinction yields a simple specification of the communication requirements, thus allowing alternative designs for the communication processes to be discussed.

The remainder of this chapter is organised as follows. In Section 3.1, the functional specification of a class of segment problems is given and a running example is introduced. Section 3.2 discusses a divide-and-conquer technique that is applicable to this class. In Section 3.3, a general parallel program scheme based on the divide-and-conquer rules is given. The complexity of the resulting programs is discussed in Section 3.4, with special attention to the communication processes. Section 3.5 shows how a related problem can be solved with a similar time complexity. Some final remarks are given in Section 3.6.

3.1 The functional specification

The class of segment problems of interest is specified by:

```
|[ p, n: int;
   f(i : 0 ≤ i < n): array of int;
  {0 < p ≤ n}
  par q : 0 ≤ q < p :
       |[ m: int;
          S.q
          {R.q : q ≠ p − 1 ∨ m = M.0.n}
       ]|
  rap
]| .
```

Where

$$M.a.b = (\otimes\, i,j : a \le i \le j \le b : F.i.j)\ .$$

$\otimes$ is a binary, associative, commutative and idempotent operator on a set and F is a function of the segments of f to that set. Segments of f are denoted by $[i,j)$, where $0 \le i \le j \le n$; $F.i.j$ denotes F applied to $[i,j)$; $F.i.i$ refers to the empty segment. Due to the commutativity and associativity of $\otimes$, any term may be split off from the quantification. The additional restriction that $\otimes$ is idempotent is made only for the sake of simplicity: in the following similar formulas can be obtained for a non-idempotent operator.

The parameterised process to be designed is $S.q$, and the parallel program consists of p processes instances obtained from S. All possible choices for the number of processes p between 1 and n are allowed. It has been left unspecified which distribution of the array f to choose, but it is the intention to obtain an *efficient* parallel program by ensuring a good load balance and low communication overhead.

In the functional specification, it is stated that process $p-1$ should provide the final answer, i.e., the value of $M.0.n$. This is not an essential restriction, as it is always possible to communicate the final answer to other processes.

Throughout this chapter the computation of the maximal segment sum is used as an example ([34] discusses the variant minimum-sum section).

Example 3.0 Casting the maximal segment sum problem into our notation gives: $\otimes = \mathbf{max}$, and for $F.i.j$ and $M.a.b$:

$$\begin{aligned} F.i.j &= (\sum h : i \leq h < j : f(h)) \\ M.a.b &= (\mathbf{max}\ i,j : a \leq i \leq j \leq b : F.i.j)\ . \end{aligned}$$

□

3.2 Divide-and-conquer rules

From the functional specification we obtain in three steps a set of divide-and-conquer rules that form the base of a parallel program scheme. First we consider the expression $M.a.c$ for $a \leq c$. It turns out that $M.a.c$ can be expressed in $M.a.b$, $M.b.c$, with b an interior point, and a continuation part specified later. The second step is to obtain a divide-and-conquer rule for this continuation part, which will be feasible if additional requirements are met. The third step is the formulation of an operator $\odot$ on four-tuples, which expresses all the computations of concern. The latter formulation is used in the next section to define a parallel program scheme.

From the definition of $M.a.c$, for $0 \leq a \leq b \leq c \leq n$, we derive the following rule:

$$\begin{aligned} & M.a.c \\ = & \quad \{\ \text{definition } M\ \} \\ & (\otimes\, i,j : a \leq i \leq j \leq c : F.i.j) \\ = & \quad \{\ \text{rewrite the range}\ \} \\ & (\otimes\, i,j : a \leq i \leq j \leq b \vee b \leq i \leq j \leq c \vee a \leq i \leq b \leq j \leq c : F.i.j) \\ = & \quad \{\ \text{range splitting, } \otimes \text{ is idempotent, definition } M\ \} \\ & M.a.b \otimes (\otimes\, i,j : a \leq i \leq b \leq j \leq c : F.i.j) \otimes M.b.c\ . \end{aligned}$$

Now, the computation of $M.a.c$ referring to segment $[a,c)$ is expressed in terms of computations for the segments $[a,b)$ and $[b,c)$, with b an interior point satisfying $a \leq b \leq c$, and a so-called continuation part: a computation on segments which crosses boundary b. Note that the expressions for $M.a.b$ and $M.b.c$ are local, in the sense that they only refer to $[a,b)$ and $[b,c)$, respectively. For the continuation part, the expression is still global, but it will appear feasible to divide it into local subexpressions if some requirements are satisfied.

Requirements.

(0) F is decomposable, i.e., there is a binary associative operator $\oplus$ such that

$$(\forall\, i,j,k : 0 \leq i \leq k \leq j \leq n : F.i.j = F.i.k \oplus F.k.j)\ .$$

(1) The following two distributivity laws hold:

$$\begin{aligned}(z \oplus x) \otimes (z \oplus y) &= z \oplus (x \otimes y)\\ (x \oplus z) \otimes (y \oplus z) &= (x \otimes y) \oplus z\ ,\end{aligned}$$

for all x, y, z.

From the first requirement it is deduced that

$$F.i.i = \text{unit}(\oplus) \text{ for all } i : 0 \leq i \leq n.$$

Each distributivity law can be obtained from the other if operator $\oplus$ is commutative.

Requirement (0) can be weakened by demanding a decomposability of the form $F.i.j = H0.i.k \oplus H1.k.j$ instead.

Generally, it is necessary to have some form of decomposability together with distributivity laws in order to rewrite the expression for the continuation part into local subexpressions. The precise conditions that allow such a rewriting, and classifications of decomposability, is a subject on its own; it is not discussed here. Our interest is a derivation of a parallel segment problem, and for convenience, we consider a model problem based on only one form of decomposability.

For the continuation part the following divide-and-conquer rule is obtained:

$$\begin{aligned}&(\otimes\, i,j : a \leq i \leq b \leq j \leq c : F.i.j)\\ =&\quad \{ \text{ see requirement (0): } F \text{ decomposable } \}\\ &(\otimes\, i,j : a \leq i \leq b \leq j \leq c : F.i.b \oplus F.b.j)\\ =&\quad \{ \text{ calculus } \}\\ &(\otimes\, i : a \leq i \leq b : (\otimes\, j : b \leq j \leq c : F.i.b \oplus F.b.j))\\ =&\quad \{ \text{ see requirement (1): distributivity, calculus } \}\\ &(\otimes\, i : a \leq i \leq b : F.i.b) \oplus (\otimes\, j : b \leq j \leq c : F.b.j)\\ =&\quad \{ \text{ definition } Tl \text{ and } Hd \}\\ &Tl.a.b \oplus Hd.b.c\ ,\end{aligned}$$

where

$$\begin{aligned}Tl.a.b &= (\otimes\, i : a \leq i \leq b : F.i.b)\\ Hd.a.b &= (\otimes\, i : a \leq i \leq b : F.a.i)\ .\end{aligned}$$

$M.a.c$ can thus be computed from the local expressions $M.a.b$, $M.b.c$, $Tl.a.b$ and $Hd.b.c$, if the requirements above are met. Divide-and-conquer rules for $Tl.a.c$ and $Hd.a.c$ can also be obtained:

$$\begin{aligned}&Tl.a.c\\ =&\quad \{ \text{ definition } Tl \}\\ &(\otimes\, i : a \leq i \leq c : F.i.c)\end{aligned}$$

$$
\begin{array}{ll}
= & \{ \text{ rewrite the range } \} \\
& (\otimes\, i : a \le i \le b \vee b \le i \le c : F.i.c) \\
= & \{ \text{ range splitting, } \otimes \text{ is idempotent, definition } Tl \} \\
& (\otimes\, i : a \le i \le b : F.i.c) \otimes Tl.b.c \\
= & \{ \text{ requirement (0), } F \text{ decomposable } \} \\
& (\otimes\, i : a \le i \le b : F.i.b \oplus F.b.c) \otimes Tl.b.c \\
= & \{ \text{ requirement (1), distributivity, definition } Tl \} \\
& (Tl.a.b \oplus F.b.c) \otimes Tl.b.c\ .
\end{array}
$$

A rule for $Hd.a.c$ is obtained in the same way.

Summarising:

$$
\begin{aligned}
M.a.c &= M.a.b \otimes (Tl.a.b \oplus Hd.b.c) \otimes M.b.c \\
Tl.a.c &= (Tl.a.b \oplus F.b.c) \otimes Tl.b.c \\
Hd.a.c &= Hd.a.b \otimes (F.a.b \oplus Hd.b.c) \\
F.a.c &= F.a.b \oplus F.b.c \\
F.a.a &= unit(\oplus)\ .
\end{aligned}
$$

For all $a, b, c : 0 \le a \le b \le c \le n$.

An alternative formulation is as follows. Define $V.a.b$ by:

$$V.a.b = (M.a.b, Tl.a.b, Hd.a.b, F.a.b)\ ,$$

then the divide-and-conquer rules can be restated as:

$$V.a.c = V.a.b \odot V.b.c\ ,$$

where operator $\odot$ is defined by:

$$
\begin{array}{l}
(s0, t0, u0, v0) \odot (s1, t1, u1, v1) \\
= \\
(s0 \otimes (t0 \oplus u1) \otimes s1, (t0 \oplus v1) \otimes t1, u0 \otimes (v0 \oplus u1), v0 \oplus v1)\ .
\end{array}
$$

In this definition, $M.0.n$ can be obtained by taking the first component of four-tuple $V.0.n$.

In the next section, it will become clear that operator $\odot$ allows for a simple formulation of the parallel program scheme. We end this section by returning to the example.

Example 3.1 In order to apply the divide-and-conquer rules to the maximal segment sum problem we only need to check if the requirements are met. Indeed, operator $\otimes = \mathbf{max}$ is associative, commutative and idempotent, the function $F.i.j$:

$$F.i.j = (\sum\, h : i \le h < j : f(h))$$

is decomposable, because $F.i.j = F.i.k + F.k.j$, for $i \le k \le j$, and hence we can use for $\oplus = +$. Furthermore, $+$ distributes over $\mathbf{max}$:

$$(x\ \mathbf{max}\ y) + z = (x + z)\ \mathbf{max}\ (y + z), \text{ for integer } x, y, z\ .$$

□

3.3 The parallel program scheme

The divide-and-conquer rules suggest splitting array f into p segments and distributing these segments across p processes. Such a distribution can be modelled by a function ℓ satisfying $\ell : [0, p+1) \rightarrow [0, n+1)$, $\ell.0 = 0$, $\ell.p = n$, and ℓ increasing. For convenience, we choose to assign segment $[\ell.q, \ell.(q+1))$ of f to process q. This is not a severe restriction, since it is always possible to renumber the process identifications.

In the previous section, it is shown that $V.0.n$ can be expressed using $\odot$; this gives the following quantification:

$$V.0.n = (\odot\ q : 0 \leq q < p : V.\ell.q.\ell.(q+1)) \ .$$

Process q can compute $V.\ell.q.\ell.(q+1)$ without interaction with other processes, because only values local to process q are involved. There are many orderings in which the operator $\odot$ can be applied to obtain $V.0.n$ (and hence $M.0.n$). For convenience, we choose to evaluate $V.0.n$ in order of increasing process number. An outline of parameterised process $S.q$ is given in Figure 3.0.

```
S.q ::
|[  x, xs: (int,int,int,int);
    S0.q
   {x = V.ℓ.q.ℓ.(q + 1)}
   ; S1.q
   {xs = V.0.ℓ.q}
   ; S2.q
   {q ≠ p − 1 ∨ m = M.0.n}
]|
```

Figure 3.0: Program text for $S.q$

We briefly sketch each process in turn.

$S0.q$ is just a sequential process; its invariant $P0.q$ is:

$$P0.q \ : \ x = V.\ell.q.k \wedge \ell.q \leq k \leq \ell.(q+1) \ .$$

The resulting program for $S0.q$ is a loop, and application of the divide-and-conquer rules with $a = \ell.q$, $b = k$, and $c = k+1$ gives:

$$V.a.(k+1) = V.a.k \odot V.k.(k+1) \ ,$$

for all a,k,q, with $a = \ell.q, \ell.q \leq k < \ell.(q+1)$.

An explicit formulation for $V.a.(k+1)$ (without $\odot$) can be obtained by using the distributivity laws and the properties:

$$M.k.(k+1) = Tl.k.(k+1) = Hd.k.(k+1) = F.k.(k+1) \otimes e \ ,$$

where e =unit($\oplus$). For example,

$$
\begin{array}{ll}
 & M.a.(k+1) \\
= & \quad \{ \text{ definition } \} \\
 & M.a.k \otimes (Tl.a.k \oplus Hd.k.(k+1)) \otimes M.k.(k+1) \\
= & \quad \{ \text{ definition } Hd.k.(k+1),\ M.k.(k+1),\ e = \text{unit}(\oplus) \ \} \\
 & M.a.k \otimes (Tl.a.k \oplus (F.k.(k+1) \otimes e)) \otimes (e \oplus (F.k.(k+1) \otimes e)) \\
= & \quad \{ \text{ distributivity laws } \} \\
 & M.a.k \otimes ((Tl.a.k \otimes e) \oplus (F.k.(k+1) \otimes e)) \\
= & \quad \{ \text{ distributivity laws } \} \\
 & M.a.k \otimes (Tl.a.k \oplus F.k.(k+1)) \otimes e \ .
\end{array}
$$

In a similar way, the following formula for $Tl.a.(k+1)$ is obtained.

$$Tl.a.(k+1) = (Tl.a.k \oplus F.k.(k+1)) \otimes e \ .$$

Combining the last two results gives:

$$M.a.(k+1) = M.a.k \otimes Tl.a.(k+1) \ .$$

Establishing the postcondition of $S1.q$ is only possible via communication; hence, assumptions have to be made about the communication network. One way is by using a *chain* network. In a chain, each process q, $0 < q$, receives from process $q-1$ the value of $V.0.\ell.q$ and computes $V.0.\ell.(q+1)$ from the received value and $V.\ell.q.\ell.(q+1)$.

The program for $S2.q$ is a simple one: process $p-1$ computes $M.0.n$ by taking the first component of four-tuple $V.0.\ell.p = xs \odot x$, the other processes perform *skip*.

In an actual implementation, operator $\odot$ on four-tuples has to be worked out. Of course, this can be done by introducing four additional variables.

Example 3.2 For the maximal segment sum problem we obtain the following program based on a *chain* communication network (cf. Figure 3.1). The rules have been further simplified by using $f(k) = F.k.(k+1)$.

As can be seen from the program, the values of $Hd.0.\ell.q$ and $F.0.\ell.q$ are not computed, since they are not necessary for the computation of $M.a.c$ with $a = 0$, $c = \ell.p = n$ in process $q = p-1$ (the boundary is $b = \ell.q$). This is caused by the fact that we have chosen a specific order, namely in order of increasing process number, to evaluate $V.0.n$ (and hence $M.0.n$). The resulting program is slightly optimised by combining the guards of $S1.q$ and $S2.q$ in one program.

Note that for $p = 1$ a sequential program is obtained, which resembles very much the sequential solution for the maximal segment sum problem. It differs only in the extra computations of $c = Hd.\ell.q.\ell.(q+1)$ and $d = F.\ell.q.\ell.(q+1)$. □

3.4 Complexity

The time complexity of the general parallel program is found by adding the time complexities of the parts $S0.q$, $S1.q$, and $S2.q$ (cf. Figure 3.0). We assume that all

```
S.q ::
[[ k, a, b, c, d, as, bs: int;
   k, a, b, c, d := ℓ.q, 0, 0, 0, 0
  {P0.q : (a, b, c, d) = V.ℓ.q.k ∧ ℓ.q ≤ k ≤ ℓ.(q + 1)}
  ; do k ≠ ℓ.(q + 1)
       → {P0.q ∧ k < ℓ.(q + 1)}
           b := (b + f(k)) max 0
         ; a := a max b
         ; c := c max (d + (f(k) max 0))
         ; d := d + f(k)
         ; k := k + 1
         {P0.q}
  od
  {(a, b, c, d) = (V.ℓ.q.ℓ.(q + 1)}
  ; if q = 0 → as, bs := 0, 0
    [] q > 0 → (q − 1)?as, bs
    fi
  {(as, bs) = (M.0.ℓ.q, Tl.0.ℓ.q)}
  ; if q = p − 1 → m := as max (bs + c) max a
    [] q < p − 1→
                 as := as max (bs + c) max a
               ; bs := (bs + d) max b
               {(as, bs) = (M.0.ℓ.(q + 1), Tl.0.ℓ.(q + 1))}
               ; (q + 1)!as, bs
    fi
]]
```

Figure 3.1: Program text for the maximal segment sum problem using a chain communication network

processes synchronise on each semicolon separating the parts, and that it takes $\mathbf{O}(1)$ time to evaluate $F.k.(k+1)$ for any k.

Each process instance $S0.q$ performs a loop with $\ell.(q+1) - \ell.q$ steps, and every step takes $\mathbf{O}(1)$ time. Hence, the time complexity of process instance $S0.q$ is $\mathbf{O}(\ell.(q+1) - \ell.q)$.

Parameterised process $S0$ depends on the data distribution specified by ℓ. A good load balance is ensured if each process instance $S0.q$ has the same amount of work to do. This suggests taking the *linear* distribution function for ℓ (see Chapter 2):

$$\ell = (\lambda\ q \cdot q * (n/p) + q \ \mathbf{min}\ (n \backslash p)) \ .$$

This yields $\mathbf{O}(\frac{n}{p})$ for the time complexity $T_{\mathrm{f}}.p.n$ of $S0.q$ and $\mathbf{O}(1)$ for the load *imbalance*.

The communication complexity $T_{\mathrm{c}}.p.n$ of communication process $S1.q$ for a *chain*

is $\mathbf{O}(p)$. Each process (except process 0) receives four values from its predecessor, performs some operations, and sends four values to its successor (except process $p-1$).

The time complexity of $S2.q$ is $\mathbf{O}(1)$.

The resulting time complexity $T.p.n$ of the parallel program scheme assuming a *chain* communication network and a *linear* distribution is:

$$T.p.n = T_{\mathrm{f}}.p.n + \alpha * T_{\mathrm{c}}.p.n = \mathbf{O}(\tfrac{n}{p}) + \mathbf{O}(p) \ .$$

Note that the first term is dominant if $p \leq \sqrt{n}$.

Reconsider the specification of communication process $S1.q$:

$$\boxed{\begin{array}{l}\{x = V.\ell.q.\ell.(q+1)\} \\ S1.q \\ \{R1.q : xs = V.0.\ell.q\}\end{array}}$$

This specification was introduced to compute $V.0.n$ (and hence $M.0.n$) in process $p-1$, but the specification is too strong for computing $m = M.0.n$. We can suffice with a weaker postcondition $R1'.q$ specifying that only process $p-1$ needs to have $V.0.n$.

$$R1'.q \ : \ q \neq p-1 \vee xs = V.0.n \ .$$

Now, $R1'.q$ can be established by computing a global combine (see Chapter 1, Section 1.5). The combine consists of p terms, each term of the form $V.\ell.q.\ell.(q+1)$. The resulting communication process is easily implemented on a hypercube network when p is a power of two, or on a tree, and has a time complexity of $\mathbf{O}(\log p)$.

The time complexity of the parallel program scheme assuming a *binary hypercube* communication network and a *linear* distribution is:

$$T.p.n = \mathbf{O}(\tfrac{n}{p}) + \mathbf{O}(\log p) \ .$$

Note that for $p = n$ the time complexity $T.p.n$ becomes $\mathbf{O}(\log p)$.

3.5 All-prefixes problem

In previous sections, we have considered a general parallel program scheme that computes the value $M.0.n$. Here, a generalisation is made to record in an array g for each i, with $0 \leq i < n$, the value of $M.0.i$ (the arrays f and g use both the *linear* distribution). We refer to this related problem as the all-prefixes problem. It is shown that this problem can be solved with a small modification to the general parallel program scheme.

The local postcondition $R'.q$ becomes:

$$R'.q \ : \ (\forall\, i : \ell.q \leq i < \ell.(q+1) : g(i) = M.0.i \ ) \ .$$

A divide-and-conquer rule is easily obtained; for all i, with $\ell.q \leq i < \ell.(q+1)$:

$$
\begin{array}{ll}
& M.0.i \\
= & \quad \{\ \ell.q \leq i, \text{ property } M\ \} \\
& M.0.\ell.q \otimes (Tl.0.\ell.q \oplus Hd.\ell.q.i) \otimes M.\ell.q.i\ .
\end{array}
$$

The all-prefixes problem can be solved if the values of $M.0.\ell.q$ and $Tl.0.\ell.q$ are computed a priori. Fortunately, they can be obtained from $V.0.\ell.q$ in process q itself! This leads to the following solution $S'.q$ for the all-prefixes problem (with the same $S0$, $S1$ as in Figure 3.0):

```
S'.q ::
[[ x, xs: (int,int,int,int);
     S0.q
   {x = V.ℓ.q.ℓ.(q + 1)}
   ; S1.q
   {xs = V.0.ℓ.q}
   ; S2'.q
   {R'.q}
]]
```

Figure 3.2: Program text for $S'.q$

Process $S2'.q$ replaces $S2.q$. In the program text (cf. Figure 3.3), the notation $x[j]$ is used to select the j^{th} component of four-tuple x.

```
S2'.q ::
[[ k, y: int;
   k, y := ℓ.q, (e, e, e, e) {y = V.ℓ.q.k}
 ; do k ≠ ℓ.(q + 1)
      → {y = V.ℓ.q.k}
        {y[0] = M.ℓ.q.k ∧ y[2] = Hd.ℓ.q.k}
        {xs[0] = M.0.ℓ.q ∧ xs[1] = Tl.0.ℓ.q}
        ; g(k) := xs[0] ⊗ (xs[1] ⊕ y[2]) ⊗ y[0]
        ; y := y ⊙ V.k.(k + 1)
        ; k := k + 1
   od
]]
```

Figure 3.3: Program text of $S2'.q$ for the all-prefixes problem

The time complexity of the parts $S0.q$ and $S2'.q$ is $\mathbf{O}(\frac{n}{p})$. The time complexity of part $S1.q$ again depends on the network used. For a *chain* network the resulting time complexity of the all-prefixes problem is $\mathbf{O}(\frac{n}{p}) + \mathbf{O}(p)$.

Process $S1.q$ computes a partial combine, and is in structure similar to computing partial sums. It is possible to give a communication process for $S1.q$ (see Section 1.5), with a time complexity of $\mathbf{O}(\log p)$.

3.6 Final remarks

In this chapter, we have presented a parallel program scheme for a class of segment problems. An instance of this class is the maximal segment sum problem. Parallel program schemes can be derived in a similar way for segment problems defined only for non-empty segments, with non-idempotent operators, and with different decompositions of F.

The crux of these parallel programs is that global expressions defined on a segment $[a, c)$ can be rewritten in terms of local expressions defined on $[a, b)$ and $[b, c)$ for $a \leq b \leq c$ and an operator $\odot$. The local expressions can be evaluated by a single process if we assign consecutive segments to processes.

The general definition in Section 3.3 of ℓ, $\ell.0 = 0$, $\ell.p = n$, and ℓ is increasing, specifies such a distribution. Here, we have used the general *linear* distribution because of its load balancing property.

The divide-and-conquer rules specify how local expressions, each computed independently by all processes, can be combined into a global expression. Some communication will be necessary to achieve this, and additional assumptions about a communication network have to be made. This allows communication processes to be designed in several ways depending on the assumed communication network, thereby influencing the resulting time complexity.

For the problem class considered here, the time complexity on a *chain* network is $\mathbf{O}(\frac{n}{p}) + \mathbf{O}(p)$ and on a *binary hypercube* network it is $\mathbf{O}(\frac{n}{p}) + \mathbf{O}(\log p)$. The related problem of all prefixes can be solved in a similar way and yields similar time complexities.

The technique we outlined is not limited to the class of segment problems presented. For example, it is also possible to consider segments which satisfy an additional property. Formally, an additional predicate X in the range of M holds, and the definition of M reads:

$$M.a.b = (\otimes\, i, j : a \leq i \leq j \leq b \wedge X.i.j : F.i.j)\ .$$

Again, the computation of $M.a.c$ can be expressed in terms of segments $[a, b)$ and $[b, c)$, $a \leq b \leq c$, and a continuation part. Further massaging of the expressions for the continuation part is necessary in order to obtain local subexpressions. A general program scheme is difficult to specify, since it depends very much on the form of predicate X and operator $\otimes$.

Assume, for instance, that predicate X can be rewritten as:

$$X.i.j = X.i.k \wedge X.k.j \wedge Y.k, \text{ for integer } i \leq k \leq j \text{ and predicate } Y\ .$$

In this form, the expression for the continuation part is given by:

$$(\otimes\, i, j : a \leq i \leq b \leq j \leq c \wedge X.i.j : F.i.j)\ ,$$

and can be computed from:

$$(\otimes\, i : a \leq i \leq b \wedge X.i.b : F.i.b) \oplus (\otimes\, j : b \leq j \leq c \wedge X.b.j : F.b.j)$$

if $Y.b$ holds.

Clearly, if $Y.b$ does not hold there is no continuation part contributing to the computation of $M.a.c$.

An example of a segment problem using predicates X and Y is the longest plateau problem [33].

In general, massaging the continuation part is a cumbersome process. A non-trivial problem is given in [56] where a parallel program is considered for the maximal length of any rightmost segment.

We have tried to obtain parallel programs for many of these segment problems using the same technique with slight modifications. Sometimes, it was necessary to rewrite the specification in such a way that it falls in the desired problem class. For instance, the maximal segment product [66] does not satisfy the requirements of the problem class outlined here. For this problem, however, it is feasible to split the problem in two subproblems, each satisfying the requirements.

The resulting parallel programs consist quite often of several computation and communication phases. For the computation phase, it is relatively easy to obtain parameterised processes, since we can reuse the techniques from sequential programming. For the communication phase, simple communication networks are sufficient. Essential in all derivations is the formulation of parameterised invariants, and the separation between the computation and communication phases.

Chapter 4

Parallel Symmetric-System Solving

Parallel algorithms for dense Cholesky factorisation and triangular system solving are developed here. The parallel Cholesky factorisation algorithm uses a Cartesian matrix distribution. For this algorithm, an analysis of the communication requirements and the work load is presented. It is shown that the *grid* distribution is a good candidate distribution, which is confirmed by timing-experiments on a 400 multi-processor system. The triangular system solver is based on the *grid* distribution as well.

4.0 Introduction

The solution of linear systems of equations is of fundamental importance in large-scale scientific computations. The development of *efficient* computer algorithms for this type of computations has become a major research topic since the beginning of the age of electronic computing. Indeed, the Atanasoff-Berry computer [59, 62] solved a linear system of up to 30 equations.

A special class of linear systems is that of the symmetric positive-definite systems (s.p.d. systems). These systems arise in many areas such as: power-network problems, discretisations of partial differential equations, and linear programming.

Characteristic of this class of linear equations is that the corresponding matrix is symmetric and positive-definite, which is equivalent to all eigenvalues of the matrix being positive [32]. The symmetry can be exploited to halve the computational effort to solve the equations.

The standard procedure to obtain a solution of a s.p.d. system

$$A\,x = b\ ,$$

where A is a given n by n s.p.d. matrix, b is the given right-hand side vector of length n, x is the unknown solution vector of length n, is as follows.

- Factorise A using Cholesky's method [76]:

$$A = L\,L^t\ ,$$

  where L is a lower triangular n by n matrix (L^t is the transpose of L).

- Solve the two corresponding triangular systems:

$$\begin{aligned} L\,y &= b\ , \\ L^t\,x &= y\ . \end{aligned}$$

The factorisation is unique if the diagonal elements of L are taken to be positive. The Cholesky method uses numerically stable diagonal elements for pivoting; there is no need for a pivot search like in LU decomposition [32].

In this chapter, we are interested in the (formal) development of an *efficient* parallel system solver implementing the solution procedure as outlined above. The construction of such a parallel solver will be based on parameterised invariants as discussed in the previous chapters. Other symmetric-system solving methods [32], for example iterative solvers, fall beyond the scope of this chapter and are not considered.

The Cholesky factorisation involves $\frac{n^3}{3} + \mathbf{O}(n^2)$ elementary operations. This is, compared to $2 * n^2 + \mathbf{O}(n)$ elementary operations required for the solution of the resulting triangular systems, the bulk of the work done. Parallelisation of only the Cholesky factorisation, however, can cause the triangular system solving part to become a bottleneck. To prevent this, parallel algorithms for both Cholesky factorisation and triangular system solving have been developed.

Most *parallel* algorithms for symmetric-system solving are based on either a row or column distribution of the matrix L [23, 27, 52]. Here, we derive a general parallel program scheme for the Cholesky factorisation, which can be instantiated with, for example, the *block*, *row*, *col*, *wrap-row* or *wrap-col* distribution. This scheme is used to obtain a new *grid*-based parallel program for the Cholesky factorisation. The resulting program has good load-balancing properties and a low communication overhead.

An outline of this chapter is as follows. The parallel program scheme for Cholesky factorisation uses a Cartesian distribution of the matrix L, and is obtained in Section 4.1. The communication requirements and the load-balancing properties are further analysed in Subsections 4.1.1 and 4.1.2. It is argued that the *grid* distribution is a good candidate distribution for the parallel Cholesky factorisation (this has been shown in the context of LU decomposition [5]; see [75] for a derivation). Timing-experiments on 400 transputers are given in Subsection 4.1.3. These experiments compare three different parallel Cholesky factorisation programs based on the *grid*, *wrap*-column and *block* distribution (Example 2.17), respectively. In Section 4.2, a derivation is given of a triangular solver assuming the *grid* distribution of L. In this way, a truly parallel symmetric-system solver is obtained that uses the same distribution. In Section 4.3, we summarise what we have achieved.

4.1 Parallel Cholesky factorisation

Given is an n by n s.p.d. matrix A. Our task is to design a parallel algorithm consisting of p processes, which computes a lower triangular matrix L satisfying

$$R \ : \ A = L\,L^t \ .$$

From postcondition R and using the symmetry of A we find:

$$(\forall\, i,j : 0 \leq j \leq i < n : A(i,j) = (\sum\, h : 0 \leq h < n : L(i,h) * L(j,h)) \ .$$

The contribution of the terms $L(j,h)$ is 0 for $h > j$, because L is lower triangular. Splitting all terms in the summation with $h = j$ and $h > j$ gives, after some calculus, a reformulated postcondition R:

$$R \ : \ (\forall\, i,j : 0 \leq j \leq i < n : L(i,j) * L(j,j) = sum.i.j.j) \ ,$$

where for all a, b, c, $0 \leq a, b < n, 0 \leq c \leq n$:

$$sum.a.b.c = A(a,b) - (\sum\, h : 0 \leq h < c : L(a,h) * L(b,h)) \ .$$

The value of $sum.i.j.j$ is determined by the first j columns of L. This indicates a sequential solution: compute the elements of L column-wise starting with the first column.

We propose to distribute L across $p = M * N$ processes using a Cartesian distribution $\mathcal{D}0 \times \mathcal{D}1$, $\mathcal{D}0 = (\delta 0, \mathbf{n}, \mathbf{M})$ and $\mathcal{D}1 = (\delta 1, \mathbf{n}, \mathbf{N})$. Each process is identified by an ordered pair (s,t), $0 \leq s < M$ and $0 \leq t < N$. One can think of the processes to be arranged as M process row ensembles of N processes each. Or, alternatively, as N process column ensembles of M processes each (cf. Figure 4.0). The distribution $\mathcal{D}0$ specifies how the n rows of a matrix are assigned to M process row ensembles; $\mathcal{D}1$ specifies how the n columns of a matrix are assigned to N process column ensembles.

A Cartesian distribution may specify that a process gets a part of the zero upper triangle of L. For example, the *block* distribution assigns zero matrix elements to each process (s,t) with $s < t$. This is not a restriction, because a process that is assigned only elements from the zero upper triangular of L is not performing any useful computations and can be ignored.

In the following, we shall omit the ranges on s and t unless stated otherwise. The notation $local.i.j$ is shorthand for:

$$\begin{aligned} local.i.j &\equiv (i,j) \in \mathcal{O}^{\delta 0 \times \delta 1}.(s,t) \\ &\equiv 0 \leq i < n \wedge \delta 0.i = s \wedge 0 \leq j < n \wedge \delta 1.j = t \ . \end{aligned}$$

Predicate $local.i.j$ holds if index pair (i,j) and a corresponding matrix element are assigned to process (s,t). A parameterised postcondition $R.s.t$ is:

$$R.s.t \ : \ (\forall\, i,j : local.i.j \wedge i \geq j : L(i,j) * L(j,j) = sum.i.j.j) \ .$$

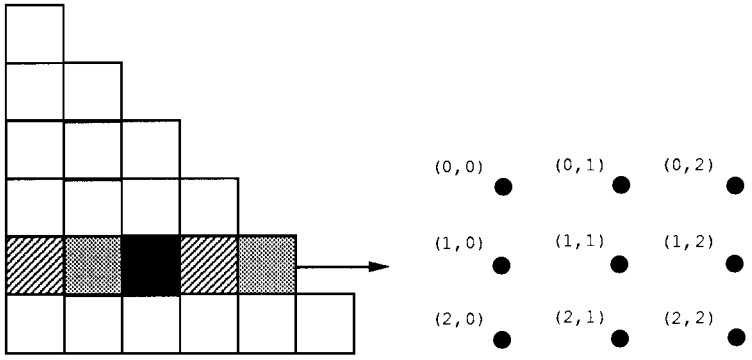


Figure 4.0: A Cartesian distribution of a 6 by 6 lower triangular matrix L across 9 processes is depicted. The fifth row of the matrix L is distributed across the second process row $(1,t)$, $0 \leq t < 3$. The grey-scales denote the assignments of matrix elements to processes: $L(4,0)$ is assigned to process $(1,0)$; $L(4,1)$ to process $(1,1)$; $L(4,2)$ to process $(1,2)$, and so forth.

This parameterised postcondition is the starting point for the derivation of a parallel Cholesky factorisation program scheme, which is given in the next subsection. The communication behaviour and load-balancing properties of the resulting parallel program scheme are discussed in the second and third subsection.

4.1.0 A derivation

We present a formal derivation of a parallel program scheme for the Cholesky factorisation. It turns out that it is possible to specify communication processes separately, and to parameterise the resulting programs in the Cartesian distribution $\mathcal{D}0 \times \mathcal{D}1$. The derivation itself is rather smooth: after having formulated the parameterised invariants, it is relatively easy to obtain the program text for the processes.

The local postcondition $R.s.t$ is used to formulate a parameterised invariant $P.s.t$. A local variable k is introduced in every process, and it is assumed that all processes have the same value of k. Inspired by a sequential solution (column-wise computation order) we propose:

$$\begin{aligned} P.s.t &: P0.s.t \wedge P1.s.t \\ P0.s.t &: 0 \leq k \leq n \\ P1.s.t &: (\forall\, i,j : local.i.j \wedge j < k \wedge i \geq j : L(i,j) * L(j,j) = sum.i.j.j)\ . \end{aligned}$$

Setting k to 0 establishes invariant $P.s.t$; the resulting program contains a loop with $k \neq n$ as guard. Invariant $P1$ expresses that the first k columns of L have been computed.

Progress is made by incrementing k; its effect on $P1$ is:

$$
\begin{array}{ll}
& P1.s.t\ (k := k+1) \\
\equiv & \quad \{\ \text{definition } P1,\ \text{substitution}\ \} \\
& (\forall\, i,j : \mathit{local.i.j} \wedge j < k+1 \wedge i \geq j : L(i,j) * L(j,j) = \mathit{sum.i.j.j}) \\
\equiv & \quad \{\ \text{range splitting } j = k,\ \text{definition } \mathit{local.i.j},\ \text{definition } P1\ \} \\
& P1.s.t \wedge (\ \delta 1.k \neq t \vee \\
& \qquad (\forall\, i : k \leq i < n \wedge \delta 0.i = s : L(i,k) * L(k,k) = \mathit{sum.i.k.k}))\ .
\end{array}
$$

From this little calculation we conclude that invariant $P1$ needs only to be restored by the M processes $(s, \delta 1.k)$. The values of $L(k,k)$ and $sum.i.k.k$, $k \leq i < n$, have to be computed before $L(i,k)$, $k < i < n$, can be computed. The value of $L(k,k)$ is only available in process $(\delta 0.k, \delta 1.k)$; therefore, some communications are needed. The values of $sum.i.k.k$ are recorded by variables in order to avoid excessive communications of elements of L. This is not sufficient, because in step $k+1$ the values of $sum.i.(k+1).(k+1)$ are needed as well. Hence, an n by n matrix X is introduced, which stores in $X(i,j)$ partial sum $sum.i.j.k$. This is expressed by $P2$:

$$
\begin{array}{rcl}
P.s.t & : & P0.s.t \wedge P1.s.t \wedge P2.s.t \\
P2.s.t & : & (\forall\, i,j : \mathit{local.i.j} \wedge k \leq j \wedge i \geq j : X(i,j) = \mathit{sum.i.j.k})\ .
\end{array}
$$

$P2$ is initialised by setting k to zero and X to A (only for the lower triangular part). Consequently, A has the same distribution as L.

The parallel program is the parallel composition of $p = M * N$ instances of parameterised process S, which is outlined in Figure 4.1.

```
S.s.t ::
[[ X(i,j : 0 ≤ i,j < n): array of real; k: int;
   k := 0
 ; for all i,j : local.i.j ∧ i ≥ j : X(i,j) := A(i,j) llа rof
 {P.s.t}
 ; do  k ≠ n →
         RestoreP1.s.t
       ; RestoreP2.s.t
       ; k := k + 1 {P.s.t}
   od
]]
```

Figure 4.1: Outline of each parameterised process S

This leaves us with the obligation to design Restore$P1$ and Restore$P2$. We start with the former (cf. Figure 4.2). If $P2$ holds all processes $(s, \delta 1.k)$ can compute an $L(i,k)$ from the values $L(k,k)$ and $X(i,k) = sum.i.k.k$. The value of $L(k,k)$ is computed by process $(\delta 0.k, \delta 1.k)$ from $X(k,k)$, and is communicated by the processes $C0.s$. Communication process $C0.s$ delivers to all processes $(s, \delta 1.k)$ a copy of $L(k,k)$.

```
Restore P1.s.t ::
[[ h: real;
   if δ0.k = s ∧ δ1.k = t
   → {X(k,k) = sum.k.k.k}
       L(k,k) := √X(k,k)
     {L(k,k)² = sum.k.k.k}
     ; h := L(k,k)
     {h = L(k,k)}
     ; C0.s
   [] δ0.k ≠ s ∧ δ1.k = t → C0.s {h = L(k,k)}
   fi
 ; if δ1.k = t
   → {h = L(k,k)}
       for all i : k+1 ≤ i < n ∧ δ0.i = s :
         {X(i,k) = sum.i.k.k}
         L(i,k) := X(i,k)/h
         {L(i,k) * L(k,k) = sum.i.k.k}
       lla rof
   fi
]]
```

Figure 4.2: Program text for Restore$P1$

Consider $P2.s.t\ (k := k+1)$:

$$
\begin{array}{ll}
 & P2.s.t\ (k := k+1) \\
= & \quad \{ \text{ definition } P2, \text{ substitution } \} \\
 & (\forall\, i,j : local.i.j \wedge k+1 \le j \wedge i \ge j : X(i,j) = sum.i.j.(k+1)) \\
= & \quad \{ \text{ definition } sum, \text{ calculus } \} \\
 & (\forall\, i,j : local.i.j \wedge k+1 \le j \wedge i \ge j : X(i,j) = sum.i.j.k - L(i,k) * L(j,k)) \ .
\end{array}
$$

This calculation reveals that $X(i,j)$ needs the values of $L(i,k)$ and $L(j,k)$ for appropriate i and j. Hence, the restoration procedure for $P2$ is a simple one: communicate the necessary values of L to each process (s,t) and update X (cf. Figure 4.3). The communication of the appropriate values of $L(i,k)$ and $L(j,k)$ is done by two communication processes $C1.s.t$ and $C2.s.t$, which use two arrays c and d to store received values.

If the statement lists of the processes are combined several optimisations are feasible, such as integration of guards and removal of matrix X. The latter can be replaced by L, since partial sums are only maintained for matrix elements $X(i,j)$, $k \le j \le i < n$. Besides the implementations of parameterised communication processes $C0$, $C1$, and $C2$, we have obtained the complete structure of the parallel program.

The resulting program is a *dense* parallel submatrix-Cholesky (an overview is given in [37]). Submatrix-Cholesky uses two basic operations: *cdiv* and *cmod*. The *cdiv.k*

Restore$P2.s.t$ ::
[[$c, d(i : 0 \leq i < n)$: array of real;
if $\delta 1.k = t$
$\rightarrow$ **for all** $i : k+1 \leq i < n \wedge \delta 0.i = s : c(i) := L(i,k)$ **lla rof**
fi
$\{\delta 1.k \neq t \vee (\forall\, i : k+1 \leq i < n \wedge \delta 0.i = s : c(i) = L(i,k))\}$
; $C1.s.t$
$\{(\forall\, i : k+1 \leq i < n \wedge \delta 0.i = s : c(i) = L(i,k))\}$
; $C2.s.t$
$\{(\forall\, j : k+1 \leq j < n \wedge \delta 1.j = t : d(j) = L(j,k))\}$
; **for all** $i, j : local.i.j \wedge k+1 \leq j \wedge i \geq j :$
$X(i,j) := X(i,j) - c(i) * d(j)$
lla rof
]]

Figure 4.3: Program text for Restore$P2$

operation scales the kth column of L by an appropriate factor. The $cmod.j.k$ operation modifies column j of L by adding a suitable multiple of column k to it (a **saxpy**). In the submatrix-Cholesky, each newly computed column k of L is used to modify *all* columns j, with $k < j$. The parallel versions of $cdiv.k$ and all $cmod.j.k$, with $k < j$, correspond precisely with the parameterised processes Restore$P1$ and Restore$P2$.

4.1.1 The communication processes

Three communication processes have been specified in the derivation. Here, each process is discussed in turn. Step k is fixed.

The specification of process $C0.s.t$ reads:

$\{\delta 0.k \neq s \vee \delta 1.k \neq t \vee h = L(k,k)\}$
$C0.s.t$
$\{\delta 1.k \neq t \vee h = L(k,k)\}$

A broadcast is specified of value $L(k,k)$ from process $(\delta 0.k, \delta 1.k)$ to the $M-1$ processes $(s, \delta 1.k)$, $0 \leq s < M \wedge s \neq \delta 0.k$. For $M = 1$, no such broadcast process is needed. Broadcasts can be easily implemented on many networks (see Section 1.3), but are expensive and should be avoided whenever possible.

The specification of communication process $C1.s.t$ reads:

$\{\delta 1.k \neq t \vee (\forall\, i : k+1 \leq i < n \wedge \delta 0.i = s : c(i) = L(i,k))\}$
$C1.s.t$
$\{(\forall\, i : k+1 \leq i < n \wedge \delta 0.i = s : c(i) = L(i,k))\}$

It is specified that each process should obtain certain matrix elements from column k of L, which is distributed across M processes $(s, \delta 1.k)$. More specifically, each process (s,t) needs exactly the values of array c that are available in process $(s, \delta 1.k)$. This results in a communication process consisting of M independent broadcasts of array parts, each broadcast to an ensemble of $N-1$ processes.

The specification of $C2.s.t$ reads:

$$\begin{array}{|l} \{(\forall\, i : k+1 \leq i < n \wedge \delta 0.i = s : c(i) = L(i,k))\} \\ C2.s.t \\ \{(\forall\, j : k+1 \leq j < n \wedge \delta 1.j = t : d(j) = L(j,k))\} \end{array}$$

Again, a communication process is needed using broadcasts. Process (s,t) needs matrix elements of column k of L specified by $\delta 1$ and k. Unfortunately, the distribution of column k is specified by $\delta 0$ and k. This means that a redistribution should take place. In the worst case the time complexity of a redistribution in step k is $\mathbf{O}(n-k-1)$, for example, when $M = 1$ (a column distribution) or $N = 1$ (a row distribution). The overall worst-case time complexity is then $\mathbf{O}(n^2)$. This is undesirable since it equals the time complexity of a *sequential* triangular system solver.

A redistribution can be avoided if $\delta 0 = \delta 1$ and $M = N$. In this case, communication process $C2$ can be implemented efficiently by M independent broadcasts: each process (t,t) broadcasts its local part of array c to the column processes (s,t), $s \neq t$. (This is also the maximal parallelism we can expect, because column k of L is distributed across M processes.)

In conclusion, a candidate distribution should satisfy $\delta 0 = \delta 1$ and $M = N$, otherwise a redistribution is necessary.

4.1.2 Candidate distributions

In this subsection, an analysis is given of the work load distribution of a process in step k. This gives an accurate expression, which is parameterised in the process number, step k, and the Cartesian distribution $\mathcal{D}0 \times \mathcal{D}1$. This expression is estimated for the particular choice $\mathcal{D}0 = \mathcal{D}1$, and it is argued that the *grid* distribution is a good candidate. We now present the analysis.

The parallel program consists of n steps with in each step some computations and communications. In step k, $0 \leq k < n$, the bulk of the computations are done by all processes in the update part of Restore$P2$; some extra computations are done by processes $(s, \delta 1.k)$ in Restore$P1$. It is assumed that all elementary operations, like subtraction, and multiplication (square roots are not counted), take the same amount of time. The computational load in step k of process (s,t), $W.s.t.k$, is obtained by counting the number of elementary operations.

$$W.s.t.k \;=\; 2 * (\sum\, j : j \in \mathcal{O}^{\delta 1}.t \cap H.(k+1) : \mid \mathcal{O}^{\delta 0}.s \cap H.j \mid)$$

$$+ \begin{cases} \mid \mathcal{O}^{\delta 0}.s \cap H.(k+1) \mid & \text{if } \delta 1.k = t \\ 0 & \text{otherwise} \end{cases} ,$$

$H.a$ denotes the set $\{i : a \leq i < n : i\}$, for $0 \leq a \leq n$.

Explanation: The first term counts the number of subtractions and multiplications in the update part of Restore$P2$ (Figure 4.3). The second term is the number of divisions in Restore$P1$ (Figure 4.2) and counts only if $\delta 1.k = t$.

Ideally, the computational load $W.s.t.k$ should be same for every process, because idle time is then 0 for every process. Good choices for $\mathcal{D}0$ and $\mathcal{D}1$ minimise the idle time as much as possible. Therefore, we consider the maximum number of operations to be performed by any process in a step. This is not a strong restriction, since in every step communication and synchronisation take place regardless of the chosen distribution.

From $W.s.t.k$ we can determine the computation complexity $T_{\Gamma}.p.n$ ($p = M * N$):

$$T_{\Gamma}.p.n = (\sum\, k : 0 \leq k < n : (\mathbf{max}\ s, t :: W.s.t.k)) .$$

It is possible to evaluate the expression above for particular distributions, but a closed formula for arbitrary distributions is unlikely to be obtained. In the following, we analyse expression $W.s.t.k$ for $\mathcal{D}0 = \mathcal{D}1$. Previously, we showed that in this case redistribution is avoided. It is assumed that $n \gg p$, and every process (s, t) has still a part of the matrix to be factorised. This assumption is not valid at the end of the factorisation process.

One can obtain the next result:

$$\begin{array}{ll} & W.s.t.k + W.t.s.k \\ \approx & \\ & 2 * \mid \mathcal{O}^{\delta 0}.s \cap H.(k+1) \mid * \mid \mathcal{O}^{\delta 1}.t \cap H.(k+1) \mid , \text{ for all } \mathcal{D}0 = \mathcal{D}1 . \end{array}$$

The proof is tedious and can be obtained by changing the summation order in the definition of W. The $\approx$ in the formula means that equality holds for processes (s, t) with $s \neq t$, $s \neq \delta 0.k$, and $t \neq \delta 1.k$, otherwise it is a lower bound. The $\mathbf{O}(\mid \mathcal{O}^{\delta 0}.s \cap H.(k+1) \mid)$ and $\mathbf{O}(\mid \mathcal{O}^{\delta 1}.t \cap H.(k+1) \mid)$ terms are neglected, because $n \gg p$.

Using this result, we derive: for all s, t, k, $\mathcal{D}0 = \mathcal{D}1$, implying $p = M^2$:

$$\begin{array}{ll} & 2 * (\mathbf{max}\ s, t :: W.s.t.k) \\ \geq & \quad \{ \text{ calculus } \} \\ & W.s.t.k + W.t.s.k \\ \approx & \quad \{ \text{ see above } \} \\ & 2 * \mid \mathcal{O}^{\delta 0}.s \cap H.(k+1) \mid * \mid \mathcal{O}^{\delta 1}.t \cap H.(k+1) \mid . \end{array}$$

Hence,

$$
\begin{array}{ll}
& (\mathbf{max}\ s,t :: W.s.t.k) \\
\geq & \\
& (\mathbf{max}\ s,t :: \mid \mathcal{O}^{\delta 0}.s \cap H.(k+1) \mid * \mid \mathcal{O}^{\delta 1}.t \cap H.(k+1) \mid) \ .
\end{array}
$$

The last expression can be further reduced:

$$
\begin{array}{ll}
& (\mathbf{max}\ s,t :: \mid \mathcal{O}^{\delta 0}.s \cap H.(k+1) \mid * \mid \mathcal{O}^{\delta 1}.t \cap H.(k+1) \mid) \\
= & \qquad \{ \text{ calculus, all counts are non-negative } \} \\
& (\mathbf{max}\ s :: \mid \mathcal{O}^{\delta 0.s} \cap H.(k+1) \mid) * (\mathbf{max}\ t :: \mid \mathcal{O}^{\delta 1.t} \cap H.(k+1) \mid) \\
\geq & \qquad \{ \text{ Example 2.11 } \} \\
& (\mathbf{max}\ s :: \mid \mathcal{O}^{grid}.s \cap H.(k+1) \mid) * (\mathbf{max}\ t :: \mid \mathcal{O}^{grid}.t \cap H.(k+1) \mid) \\
= & \qquad \{ \text{ calculus } \} \\
& ((n-k-1+M-1)/M)^2 \ .
\end{array}
$$

Thus, we arrive at the following result.

$$(\mathbf{max}\ s,t :: W.s.t.k) \geq ((n-k-1+M-1)/M)^2 \ , \text{ for } \mathcal{D}0 = \mathcal{D}1 \ .$$

The lower bound is attained for $\mathcal{D}0 = \mathcal{D}1 = wrap$. In words: in every step k, there is a process involving at least $((n-k-1+M-1)/M)^2$ operations on its local part of the $n-k-1$ by $n-k-1$ triangular submatrix X, for *every* Cartesian distribution that assigns rows the same way as columns.

The time complexity of the *grid*-based submatrix-Cholesky is:

$$\frac{n^3}{3*p} + \mathbf{O}(\tfrac{n^2}{\sqrt{p}}) \ .$$

The time complexity of the *wrap*-column-based submatrix-Cholesky is:

$$\frac{n^3}{3*p} + \mathbf{O}(n^2) \ .$$

These results can be obtained from a precise count of the expression $W.s.t.k$ for each distribution. Most parallel Cholesky factorisation algorithms use the *wrap*-column distribution, see for example [27]. This distribution has two disadvantages: a non-scalable quadratic term in both the communication and computation complexity.

From these arguments we conclude that the $grid = wrap^2$ distribution is a suitable candidate for ensuring load balance and avoiding redistribution in the communication processes.

4.1.3 Experiments

A number of experiments were performed on a 400 transputer network. Three implementations based on the parallel Cholesky factorisation scheme were obtained by using the distributions *grid*, *wrap*-column, and *block*. Each implementation was

recoded in order to make use of distribution specific properties. For example, no communication processes $C0$ and $C1$ are needed for the *wrap*-column distribution.

In the following, we give timing-results of two experiments. In both experiments, the programs are called *grid*, *wrap* and *block* according to distribution used. The purpose of these experiments is to compare the performance of the different programs. The absolute performance, and related speed-up, is not investigated.

The programs execute on a square 20 by 20 mesh of transputers; process (s,t) is mapped one-to-one to processor (transputer) (s,t) in the mesh. The programs are implemented in transputer Pascal [57], and all computations are done in single precision (32-bit). The communication processes use directly the communication links of each processor in the mesh.

In the first experiment, we measured the execution time of the different parallel Cholesky factorisation programs on 400 processors using different matrices sizes. The dimensions of the matrix A range from 400 up to 1200 with steps of 200. For ease of comparison, the timings of *block* are corrected with a factor $\frac{2*M^2}{M^2+M} = \frac{40}{21}$ $(M = 20, p = M^2)$ in order to get a time estimate for the program on 400 processors. The reason is that the *block* distribution of a lower triangular matrix assigns no matrix elements to processes (s,t) with $s < t$. This results in a computation with effectively only $(M^2 + M)/2 = 210$ processors involved. The correction with a factor for the *block* program blurs the comparison. Fair comparison of different implementations is, indeed, a difficult task.

n	*grid*	*wrap*	*block*
400	0.6	2.0	1.0
600	1.3	4.6	2.7
800	2.6	8.4	5.6
1000	4.6	13.7	10.3
1200	7.3	20.4	16.9

Table 4.0: Execution times (in seconds) of three parallel Cholesky factorisation programs, *grid*, *wrap* and *block*, on a square mesh of 400 transputers

From Table 4.0 can be concluded that the *grid* program is superior to the *wrap* and *block* program by at least a factor of two in speed. The differences between the numbers can be understood by considering the time complexities of the programs *grid*, *wrap*, and *block*:

$$\frac{n^3}{3*p} + \mathbf{O}(\tfrac{n^2}{\sqrt{p}}) \ , \quad \frac{n^3}{3*p} + \mathbf{O}(n^2) \ , \quad \frac{n^3}{p} + \mathbf{O}(\tfrac{n^2}{\sqrt{p}}) \ ,$$

respectively. The *grid* and *wrap* distribution yield the same first-order term in the complexity results. Surprisingly, the first-order term in the complexity results for the *block* distribution is three times larger, due to load imbalance. The *block* distribution assigns approximately square submatrices of L to processes. Until the end of the Cholesky factorisation, there is a process that has to perform updates for its local

submatrix, thus causing a large load imbalance. In contrast to a *grid*-based program in which each process performs updates for a triangular submatrix whose dimensions decrease in every step.

There is at least a factor of two difference in the timing-results for *grid* and *wrap*. This is mainly due to the dominance of the second-order term in the complexity results for *wrap*. Another fact that contributes is the small number of columns (1-3) assigned to each process in the *wrap* program. This limits the range of applicability for the *wrap* program to $p \ll n$.

The *block* program shows in these experiments a competitive behaviour with the *wrap* program. This is only polish if we consider the results of second experiment in which the size of matrix A is increased.

n	*grid*	*wrap*	*block*
1500	13.6	34.6	33.3
2000	29.4	66.5	75.1
2500	54.3	111.7	141.2
3000	90.1	171.9	237.5

Table 4.1: Execution times (in seconds) of the three parallel Cholesky factorisation programs for large matrix sizes

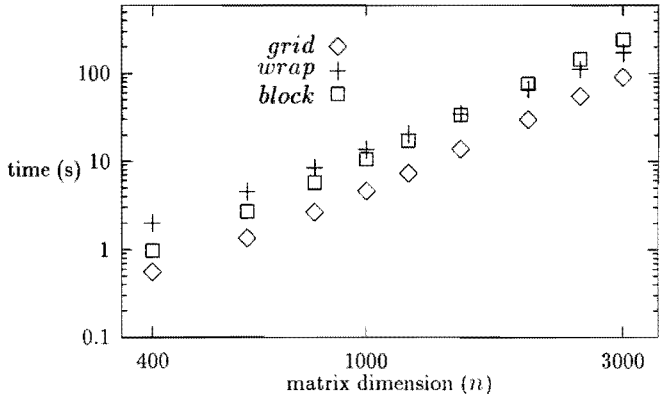


Figure 4.4: Log-log plot of all timing-results on a square mesh of 400 transputers

From Table 4.1 can be concluded that *grid* maintains its superiority. The differences in the timings between *grid* and *wrap* are smaller with increasing matrix sizes. We also find that, for $n \geq 2000$, *block* becomes slower than *wrap*. Asymptotically, it is expected that timing-results for the *grid* and *wrap* converge, since they have same first-order term in the complexity results; for the *block* it is expected that it will lie a factor of three higher. This is indeed observed.

In Figure 4.4, the timing-results of both experiments are combined into one single plot.

4.2 Parallel triangular system solving

The solution of the triangular system

$$L\,x = b$$

is sequentially an easy task due to the triangularity of L. In the parallel case the situation is different, because of the restriction that $x(j)$ can only be solved if all $x(i)$, $0 \leq i < j$, are computed. In the past many efforts have been spent in developing efficient parallel triangular system solvers based on either a row or a column distribution of the matrix [19, 38, 52]. Here, we present a formal derivation of a parallel *grid*-based triangular system solver, which is named the QWERTY algorithm. This solver can be used in combination with a parallel *grid*-based Cholesky factorisation algorithm to form a powerful symmetric-system solver.

The motivation for giving a derivation of the QWERTY algorithm was to add a formal correctness proof to the algorithm presented by R.H. Bisseling in 1988 at the Shell Conference on Parallel Computing in Amsterdam. A detailed explanation of this algorithm and timing-experiments on transputer meshes have been presented in [6].

At the time, we felt that a *formal* derivation of the QWERTY algorithm would be a challenging test-case for the use of parameterised invariants. This is essentially what is presented in [55], and is presented here in adapted form.

An outline of this section is as follows. In the first subsection a derivation using parameterised invariants is given. In the second subsection the complexity of the triangular system solver on a complete network is discussed.

4.2.0 A derivation

The problem is:

$$R \;:\; L\,x = b\;,$$

where L is an n by n lower triangular matrix distributed across $p = M * N$ processes with $M = N$ using the *grid* distribution. For the sake of simplicity we assume $n \backslash M = 0$. The vectors x and b of length n are distributed like the main diagonal of L, i.e., process (s, s) is assigned all $x(i)$ and $b(i)$ with $i \backslash M = s$.

As in the previous sections, postcondition R forms the starting point for obtaining parameterised invariants.

The postcondition is rewritten using the lower triangularity of L:

$$R \;:\; (\forall\, i : 0 \leq i < n : L(i,i) * x(i) = b(i) - sum.i)\;,$$

where

$$sum.i = (\sum\, j : 0 \leq j < i : L(i,j) * x(j))\;.$$

For convenience, we assume that $L(i,i) = 1$, for all $i : 0 \leq i < n$. This avoids expressions $L(i,i) * x(i)$ in the derivation. From postcondition R':

$$R' \ : \ (\forall\, i : 0 \leq i < n : x(i) = b(i) - sum.i) \ ,$$

we can easily satisfy postcondition R by dividing each $x(i)$ by $L(i,i)$.

The derivation is continued from postcondition R'. The parameterised postcondition $R.s.t$ is (taking the distribution of x and b into account):

$$R.s.t \ : \ s \neq t \vee (\forall\, i : 0 \leq i < n \wedge i \backslash M = s : x(i) = b(i) - sum.i) \ .$$

Generalising the parameterised postcondition gives a parameterised invariant. We propose two invariants: $P0$ and $P1$.

$$\begin{aligned} P0.s.t \ &: \ 0 \leq k \leq n \wedge k \backslash M = 0 \\ P1.s.t \ &: \ s \neq t \vee (\forall\, i : 0 \leq i < k \wedge i \backslash M = s : x(i) = b(i) - sum.i) \ . \end{aligned}$$

$P1$ is derived from R by replacing constant n by variable k, which is local to process (s,t). As a consequence of $P0$, process (s,t) contains a loop with initialisation $k := 0$, guard $k \neq n$, and increment $k + M$. The rabbit that pops up out of the hat is the step size M instead of the usual increment by one. This is, indeed, a key point in the construction of the parallel triangular solver, and can be understood if we carry the derivation a bit further.

Consider $P1.s.t\ (k := k + M)$:

$$\begin{aligned} & P1.s.t\ (k := k + M) \\ = \quad & \{ \text{ definition } P1, \text{ substitution } \} \\ & s \neq t \vee (\forall\, i : 0 \leq i < k + M \wedge i \backslash M = s : x(i) = b(i) - sum.i) \\ = \quad & \{ \text{ range splitting, } i = k + s,\ k \backslash M = 0 \ \} \\ & s \neq t \vee (\ (\forall\, i : 0 \leq i < k \wedge i \backslash M = s : x(i) = b(i) - sum.i) \wedge \\ & \qquad x(k+s) = b(k+s) - sum.(k+s)) \\ = \quad & \{ \text{ calculus, definition } P1 \ \} \\ & P1.s.t \wedge (s \neq t \vee x(k+s) = b(k+s) - sum.(k+s)) \ . \end{aligned}$$

The value $x(k + s)$ has to be calculated by process (s,t) with $s = t$. The term $sum.(k + s)$ can be expressed as a sum of partial sums $psum$, such that each partial sum contains only elements of L that are local to process (s,t); this obviates the need to communicate elements of L during the computation of a partial sum. Of course, each partial sum itself has to be added globally, i.e., via a communication process. The construction of the algorithm is driven by avoiding communication of matrix elements of L as much as possible.

Rewriting $sum.(k + s)$ gives:

$$sum.(k + s)$$

$$
\begin{array}{ll}
= & \{ \text{ definition } \} \\
& (\sum i : 0 \le i < k+s : L(k+s,i) * x(i)) \\
= & \{ \text{ calculus } \} \\
& (\sum t : 0 \le t < M : (\sum j : 0 \le j < k+s \wedge j \backslash M = t : L(k+s,j) * x(j))) \\
= & \{ \text{ range splitting } t < s \text{ and } t \ge s \ \} \\
& (\sum t : 0 \le t < s : (\sum j : 0 \le j < k+s \wedge j \backslash M = t : L(k+s,j) * x(j))) + \\
& (\sum t : s \le t < M : (\sum j : 0 \le j < k+s \wedge j \backslash M = t : L(k+s,j) * x(j))) \\
= & \{ \text{ calculus, definition } psum \ \} \\
& (\sum t : 0 \le t < s : psum.k.t.(k+s) + L(k+s,k+t) * x(k+t)) + \\
& (\sum t : s \le t < M : psum.k.t.(k+s)) \ ,
\end{array}
$$

where for all a,r,i with $0 \le a \le n$, $0 \le i < n$, and $0 \le r < M$:

$$psum.a.r.i = (\sum j : 0 \le j < a \wedge j \backslash M = r : L(i,j) * x(j)) \ .$$

In this form, the sequential order between the values $x(k+s)$ becomes clearly visible. The value of $x(k+s)$ can be computed from $sum.(k+s)$ which itself can be computed from a number of *psum* values and all $x(k+t)$ with $t < s$. The idea is to compute the values of *psum* locally by each process and combine them in a communication process in order to compute $sum.(k+s)$. The partial sum $psum.k.t.(k+s)$ contains matrix elements $L(k+s,j)$, with $0 \le j < k \wedge j \backslash M = t$, that are local in process (s,t). In step k, the values of $x(j)$, with $j < k$ are known, and hence $psum.k.t.(k+s)$ can be computed. The products $L(k+s,k+t) * x(k+t)$, $t < s$, in the summation can only be computed if the values $x(j)$, $k \le j < k+s$, are available, i.e., when all invariants $P1.(j \backslash M).(j \backslash M)$ $(k := k+M)$ hold. It is clear that $P1.s.s$ can be restored in the order $s = 0, 1, \ldots, M-1$. As we will see later, an ordering on the invariants can be given formally using a ranking function.

The computation of *psum* can be kept invariant by introducing in each process a variable w. Suggesting an invariant of the form:

$$P2'.s.t \ : \ w = psum.k.t.(k+s) \ .$$

It can be concluded from $P2'.s.t$ $(k := k+M)$ that a single variable w is not sufficient. Therefore, in each process an array of variables w is introduced, which maintains for each row i, with $i \ge k+s$, a partial sum *psum*. This is expressed by $P2$ [0]:

$$P2.s.t \ : \ (\forall \, i : k+s \le i < n \wedge i \backslash M = s : w(i,t) = psum.k.t.i) \ .$$

For notational purposes we introduce an extra index t in w; this allows us to make a distinction between the local variables w in different processes. As a consequence, w can be seen as a large n by M matrix distributed according to $(wrap, \mathbf{n}, \mathbf{M}) \times (identity, \mathbf{M}, \mathbf{M})$.

An outline of the QWERTY algorithm is given in Figure 4.5. In the program text array w is set to zero; a loop is shown calling the parameterised processes *Restore*P1 and *Restore*P2.

[0]Invariant $P2$ is weaker than presented in the original paper [55] and was suggested by [24].

$S.s.t$ ::
[[$w(i,t : 0 \leq i < n, 0 \leq t < M)$: matrix of real; k: int;
  $k := 0$
 ; **for all** $i : 0 \leq i < n \wedge i \backslash M = s : w(i,t) := 0$ **lla rof**
 $\{P.s.t : P0.s.t \wedge P1.s.t \wedge P2.s.t\}$
 ; **do** $k \neq n \rightarrow$
  Restore$P1.s.t$ $\{P1.s.t\ (k := k + M)\}$
  ; Restore$P2.s.t$ $\{P2.s.t\ (k := k + M)\}$
  ; $k := k + M$ $\{P.s.t\}$
 od
]]

Figure 4.5: Outline of each parameterised process $S.s.t$ of QWERTY

Parameterised process Restore$P1$ has the following specification:

$\{P0.s.t \wedge P1.s.t \wedge P2.s.t\}$
$\{w(k+s,t) = psum.k.t.(k+s)\}$
Restore$P1.s.t$
$\{s \neq t \vee x(k+s) = b(k+s) - sum.(k+s)\}$
$\{P1.s.t\ (k := k + M)\}$

The program text for Restore$P1$ is easily obtained by using the derived rule for $sum.(k+s)$:

$sum.(k+s)$
=
$(\sum t : 0 \leq t < s : psum.k.t.(k+s) + L(k+s, k+t) * x(k+t)) +$
$(\sum t : s \leq t < M : psum.k.t.(k+s))$.

The diagonal processes (s,s) will each compute $x(k+s)$ from $sum.(k+s)$ as follows. The processes (s,t) with $s < t$ communicate their $psum.k.t.(k+s)$ to process (s,s). The processes (s,t) with $s > t$ have to obtain first the value of $x(k+t)$ from process (t,t) before they can communicate the value of $psum.k.t.(k+s) + L(k+s, k+t) * x(k+t)$ to process (s,s). The program text for this complicated communication process is given in Figure 4.6. It is assumed that all communication is done using a complete communication network. In the program text an array a is used to store received messages. Now, we focus on Restore$P2$. Consider $P2.s.t\ (k := k + M)$:

$P2.s.t\ (k := k + M)$
= { substitution }
$(\forall i : k + s + M \leq i < n \wedge i \backslash M = s : w(i,t) = psum.(k+M).t.i)$
= { definition $psum$, calculus }
$(\forall i : k + s + M \leq i < n \wedge i \backslash M = s : w(i,t) = psum.k.t.i + L(i, k+t) * x(k+t))$.

$P2$ can easily be restored using the value of $x(k+t)$. For processes (s,t) with $s > t$, the value of $x(k+t)$ is available in local variable y on account of Restore$P1$. For processes (s,t) with $s < t$, the value of $x(k+t)$ is communicated by process (t,t). The resulting program is given in Figure 4.7.

```
Restore P1::
[[ a(i : 0 ≤ i < M): array of real; y: real;
  if s < t → {w(k+s,t) = psum.k.t.(k+s)} (s,s)!w(k+s,t)
  [] s = t →  par u : t < u < M :
                  (s,u)?a(u)
                  {a(u) = psum.k.u.(k+s)}
              rap
            ; par u : 0 ≤ u < t :
                  (s,u)?a(u)
                  {a(u) = psum.k.u.(k+s) + L(k+s,k+u) * x(k+u)}
              rap
            ; a(t) := w(k+s,t)
            {a(t) = psum.k.t.(k+s)}
            ; x(k+s) := b(k+s)
            ; for all u : 0 ≤ u < M : x(k+s) := x(k+s) − a(u) lla rof
            {x(k+s) = b(k+s) − sum.(k+s)}
            ; par u : s < u < M : (u,t)!x(k+s) rap
  [] s > t →  (t,t)?y
            {y = x(k+t) ∧ w(k+s,t) = psum.k.t.(k+s)}
            ; (s,s)!w(k+s,t) + L(k+s,k+t) * y
  fi
]]
```

Figure 4.6: Program text for Restore$P1$

```
Restore P2::
  if s < t → (t,t)?y {y = x(k+t)}
  [] s = t →  y := x(k+t)
              {y = x(k+t)}
              ; par u : 0 ≤ u < s : (u,t)!y rap
  fi
{y = x(k+t)}
; for all i : k+s+M ≤ i < n ∧ i\M = s :
        w(i,t) := w(i,t) + L(i,k+t) * y
  lla rof
```

Figure 4.7: Program text for Restore$P2$

The restoration procedures for invariants $P1.s.s$ and $P2.s.t$ are based on the derivations of $P1.s.s$ $(k := k + M)$ and $P2.s.t$ $(k := k + M)$ (the invariants $P0.s.t$ and $P1.s.t$, with $s \neq t$, are trivially maintained). Since the restoration procedure of an invariant assumes the validity of other invariants (with different s, t, and k), it is not *a priori* clear that there exists an order in which the invariants can be established. The situation in which there is no such order is called *computational deadlock*, to be distinguished from *communication deadlock*, which may occur in an actual implementation. In the following, we define a ranking function on the invariants; its existence encapsulates the absence of *computational deadlock*. Before giving such a ranking function, we define some notions.

Definition 4.0 (Inv) The finite set of invariants Inv is given by:

$$\begin{aligned} Inv \;=\; & \{s,k : 0 \leq s < M \wedge 0 \leq k \leq n \wedge k \backslash M = 0 : (1,s,s,k)\} \cup \\ & \{s,t,k : 0 \leq s,t < M \wedge 0 \leq k \leq n \wedge k \backslash M = 0 : (2,s,t,k)\} \; . \end{aligned}$$

There is an obvious one-to-one correspondence between the four-tuples of set Inv and invariants $P1.s.s$ and $P2.s.t$ in step k. □

Definition 4.1 ($\prec$) We define a relation $\prec$ on $Inv \times Inv$ with the following meaning:

$$I0 \prec I1 \equiv I0 \text{ must hold before } I1 \text{ can hold} \; ,$$

for all $I0 \neq I1 \in Inv$.

The definition of the above relation is meaningful, because of the correspondence between Inv and the parameterised invariants $P1$ and $P2$. □

Definition 4.2 ($\overset{+}{\prec}$) $\overset{+}{\prec}$ is the transitive non-reflexive closure of $\prec$. □

The elements of the relation $\prec$ are:

$$\begin{aligned} (1,s,s,k) &\prec (1,s,s,k+M) \\ (2,s,t,k) &\prec (2,s,t,k+M) \\ (1,t,t,k) &\prec (2,s,t,k+M) \\ (2,s,t,k) &\prec (1,s,s,k+M) \\ (1,t,t,k+M) &\prec (1,s,s,k+M) \text{ for } t < s \end{aligned}$$

for all $s,t,k : 0 \leq s,t < M \wedge 0 \leq k < n \wedge k \backslash M = 0$.

This follows from the derivation. The first four definitions of the elements of $\prec$ are fairly standard. The fifth one follows from the rewrite rule from $sum.(k+s)$ (see page 74).

Initially, all invariants $P1.s.t$ (k) and $P2.s.t$ (k) with $k = 0$ hold, i.e., are established. If a path in Inv is followed, starting from the initial invariants, then we do not wish to encounter cycles, since this implies that it is impossible to find an order in which the invariants can be maintained. Computational deadlock occurs if such an order does not exist.

Definition 4.3 (No computational deadlock)

$$
\begin{array}{ll}
& \text{No computational deadlock} \\
\equiv & \\
& (Inv, \overset{+}{\prec}) \text{ is irreflexive} \\
\equiv & \quad \{ \text{ definition irreflexive } \} \\
& (\forall\, I : I \in Inv : \neg(I \overset{+}{\prec} I)) \; .
\end{array}
$$

□

One way to proof irreflexivity in a relational system $(Inv, \overset{+}{\prec})$ is by showing the existence of a so-called ranking function. Actually, we have to prove that $(Inv, \overset{+}{\prec})$ is a strict order [11], i.e., a relational system that is transitive and irreflexive.

Definition 4.4 (ranking function) A ranking function r is a function from Inv to the natural numbers such that:

$$(\forall\, I0, I1 : I0, I1 \in Inv \wedge I0 \overset{+}{\prec} I1 : r.I0 < r.I1) \; .$$

□

For the QWERTY algorithm, the following ranking function r can be given.

$$
\begin{aligned}
r.(1, s, s, k) &= 2 * k + 2 * s + 1 \\
r.(2, s, t, k) &= 2 * k + 2 * s + 2 \; .
\end{aligned}
$$

It can easily be verified that $r.I0 < r.I1$, for all $I0, I1 \in Inv$ with $I0 \prec I1$. This proves that r is indeed a ranking function.

The problem of computational deadlock has not been addressed before, since it has been relatively easy to find an order in which the invariants can be restored. In the QWERTY algorithm, the situation is different. In order to demonstrate absence of computational deadlock, we used ranking functions. Of course, this technique can be applied to other problems.

4.2.1 Complexity of the triangular solver

The QWERTY algorithm is derived under the assumption of a complete network for the communication processes. It is assumed that communication takes α time units, and communications within a **par**-statement are counted as a single communication.

The complexity of the parallel program is obtained by summing the complexities of processes Restore$P1$ and Restore$P2$ in each step k.

We start with the latter. The bulk of the computational work in step k is in the **for all**-statement of Restore$P2$ (cf. Figure 4.7):

$$
\begin{array}{ll}
& 2* \mid \{i : k + s + M \leq i < n \wedge i \backslash M = s : i\} \mid \\
= & \quad \{ \text{ calculus } \} \\
& 2 * ((n - k)/M - 1) \; .
\end{array}
$$

The resulting communication complexity is 1, since one single value is broadcasted in parallel. The total complexity for Restore$P2$ is:

$$T_{P2}.M.n.k = 2 * ((n-k)/M - 1) + \alpha \ .$$

The time complexity of Restore$P1$ is obtained by a careful analysis of the critical path of the data flow. In order to obtain a low complexity, the program should be transformed by rewriting the program text of Restore$P1$ for processes (s,t) with $s = t$ (cf. Figure 4.6).

The resulting implementation of Restore$P1$ consists of two phases:

In the first phase, all processes (s,t) with $s \leq t$ are active. The processes (s,s) perform the initialisation $x(k+s) := b(k+s)$. The processes (s,t), with $s < t$, send their value of w in parallel to process (s,s) in α time. These values and the local value of w of process (s,s) are subtracted from $x(k+s)$ in at most M time units. The total complexity of the first phase of Restore$P1$ in step k is $M + \alpha$.

In the second phase, all processes (s,t) with $s \geq t$ are active. Each active process column (s,t), with $s > t$, receives the value of $x(k+t)$ in time α, which can be used to compute $w(k+s,t) + L(k+s,k+t) * y$ in two time units. The value of the last expression is received in $a(t)$ by process (s,s) in time α, which *immediately* subtracts $a(t)$ from $x(k+s)$ in one time unit (instead of first collecting the values and then subtracting, which would cause a delay along the critical path of the data flow).

The critical path of second phase is the data flow from process $(0,0)$, to $(1,0)$, to $(1,1)$, $\ldots$, to $(M-1, M-1)$. A process (s,s) on this path receives a value from process $(s,s-1)$, subtracts it from the current value of $x(k+s)$, and sends $x(k+s)$ to process $(s+1,s)$ (and to the other processes in the same active process column). This process in turn uses the received value to compute $w(k+s,s+1) + L(k+s,k+s+1) * y$, which is sent to process $(s+1,s+1)$. The time of the critical path is at most $3 * (M-1) + 2 * (M-1) * \alpha$, and this is the complexity of the second phase of Restore$P1$.

Adding the time complexities of the first and second phase in Restore$P1$ gives:

$$T_{P1}.M.n.k = 4 * M - 3 + (2 * M - 1) * \alpha \ .$$

The total time complexity for the QWERTY algorithm is:

$$\begin{array}{ll} & T.p.n \\ = & \quad \{ \text{ definition } \} \\ & (\sum\, k : 0 \leq k < n \wedge k \backslash M = 0 : T_{P1}.M.n.k + T_{P2}.M.n.k) \\ = & \quad \{ \text{ definition } T_{P1} \text{ and } T_{P2}, \text{ calculus } \} \\ & (\sum\, k : 0 \leq k < n \wedge k \backslash M = 0 : 2 * ((n-k)/M) + 4 * M - 5 + 2 * M * \alpha) \\ = & \quad \{ \text{ calculus, } p = M^2 \ \} \\ & \frac{n^2}{p} - 4 * \frac{n}{\sqrt{p}} + (2 * \alpha + 4) * n \ . \end{array}$$

The results are valid on a complete network ($p > 1$). Similar counts can be obtained for a square mesh communication network; this has also been verified experimentally [6].

4.3 Final remarks

In this chapter, we demonstrated the use of parameterised invariants on a non-trivial problem: a parallel symmetric-system solver. The solver consists of two parts: Cholesky factorisation and triangular system solving.

We showed that a parallel Cholesky factorisation algorithm can be derived formally using a Cartesian matrix distribution. Communication and computation aspects are easily separated in the derivation. This allows us to analyse work load distribution and communication overhead.

It has been argued that the $wrap^2 = grid$ distribution is a good candidate distribution for the Cholesky factorisation. This is confirmed by experiments on a 400 multi-processor system. The *grid* distribution is used in the precondition of the parallel triangular system solver, which is named the QWERTY algorithm.

Some lessons have been learned from the QWERTY derivation:

- Avoidance of communication of matrix elements is the driving force behind the derivation. In general, avoidance of communication is a basic principle in parallel program construction.
- Typical distribution properties, like a step size of M in a loop, are surprising in a derivation. In that sense, derivations are a trial and error process. In a presentation, like the one here, the trials are usually omitted.
- The standard way of constructing sequential algorithms is: introduce an invariant, some calculus, and then strengthen the invariant by new ones, etc. In the parallel case, this may pose a problem, since invariants are parameterised. Many more orderings between invariants play a rôle, thus complicating the correctness concerns quite a bit. In the QWERTY derivation, the concept of computational deadlock is encountered, and a non-standard ordering between the parameterised invariants is needed. Ranking functions are necessary to proof absence of computational deadlock.

The QWERTY derivation results in a parallel program with a low complexity and an asymptotically maximal speed-up. Hence, triangular system solving is not the bottleneck in a parallel symmetric-system solver. The complexity of such a solver is dominated by the complexity of a parallel *grid*-based Cholesky factorisation program, which is:

$$T.p.n = \frac{n^3}{3 * p} + \mathrm{O}(\tfrac{n^2}{\sqrt{p}}) \ .$$

The *grid* distribution is the key to a highly efficient parallel symmetric-system solver.

Chapter 5

Parallel Sparse Cholesky Factorisation

Two parallel algorithms are presented for the Cholesky factorisation of a *sparse* matrix. Both algorithms are based on a submatrix-Cholesky algorithm using the *grid* distribution for the non-zeros of the matrix. The fastest version uses multiple-rank updates. In this way, natural parallelism is exploited, which can be obtained by pre-permuting the matrix according to a *layered-defoliation* strategy of the corresponding elimination tree.

5.0 Introduction

There are many definitions of a sparse matrix. One is [73]:

> *A matrix is called sparse if the overwhelming majority of matrix elements are zero.*

Another one is [16]:

> *Generally, we say that a matrix is sparse if there is an advantage in exploiting its zeros.*

Most practical problems involve sparse matrices, for instance, the calculation of stiffness properties of buildings and the optimisation of refinery and scheduling operations. Sparse matrices do not only occur in numerical mathematics, but also in graph theory [36] and many other fields.

A large class of sparse matrix problems requires the solution of a *sparse* linear system of equations. Exploiting sparsity in such a system typically results in a considerable reduction of memory requirements and computation time. Zero matrix elements need not to be stored and floating-point operations involving them are usually made redundant.

Here, we develop new parallel algorithm for the Cholesky factorisation of a *sparse* symmetric positive-definite matrix A. The Cholesky factorisation $A = L\,L^t$, where L is a lower n by n triangular matrix, forms a key component in a parallel linear programming solver [3].

In our opinion, the parallelisation of sparse algorithms is a challenging activity. The challenge lies in the conflicting nature of sparse algorithms that can be summarised by the statement: 'a lot of little work'. Sparse algorithms try to avoid unnecessary work, but the total amount of work can be high, thus making it worthwhile to speed up the computation by parallelisation.

The remainder of this chapter is organised as follows. In Section 5.1, we give some background on sparse Cholesky factorisation; for an extensive treatment see [16, 29]. In Section 5.2, we briefly review parallel Cholesky factorisation algorithms. In Subsection 5.3.0, the properties of the *grid* distribution with respect to sparse Cholesky factorisation are discussed. Subsection 5.3.1 presents a parallel sparse Cholesky algorithm. A complexity analysis is given in Subsection 5.3.2. Timing results for the parallel rank-1 algorithm are given in Subsection 5.3.3. Section 5.4 describes an improvement of this algorithm by performing multiple-rank updates, which are obtained by a *layered-defoliation strategy* of the elimination tree. These subjects are discussed in Subsections 5.4.0, 5.4.1, and 5.4.2. Timing results for the parallel multiple-rank update algorithm are given in Subsection 5.4.4. Section 5.5 summarises our contribution.

A final remark is made about the presentation in this chapter. In contrast to the previous chapters, the emphasis will not be on a formal derivation, but on the final algorithms. However, the formally derived algorithm for parallel *dense* Cholesky factorisation (see Chapter 4) is used as a basis for the sparse algorithms.

5.1 Background

An important difference between sparse and dense Cholesky factorisation algorithms is the rôle of zero matrix elements, which are numerous in the sparse case. During a *sparse* factorisation of A most zero matrix elements of A are also zero in the Cholesky factor L, i.e., only some zero matrix elements in A become non-zero in L. The number of non-zeros in the Cholesky factor, the *fill*, determines the computation complexity [29]. The *fill* can be influenced by symmetrically permuting rows and columns of A. Mathematically, the following systems are equivalent:

$$\begin{aligned} A\,x &= b \\ PAP^t\,y &= Pb \quad \text{and} \quad P^t\,y = x\ , \end{aligned}$$

where P is an n by n permutation matrix. The non-zero patterns of the Cholesky factors of A and PAP^t may differ considerably (cf. Figures 5.0 and 5.1). Often, P can be chosen such that the Cholesky factor of PAP^t has less *fill* than the Cholesky

factor of A. Dense Cholesky factorisation algorithms lack this degree of freedom; the computation complexity is independent of any choice for P.

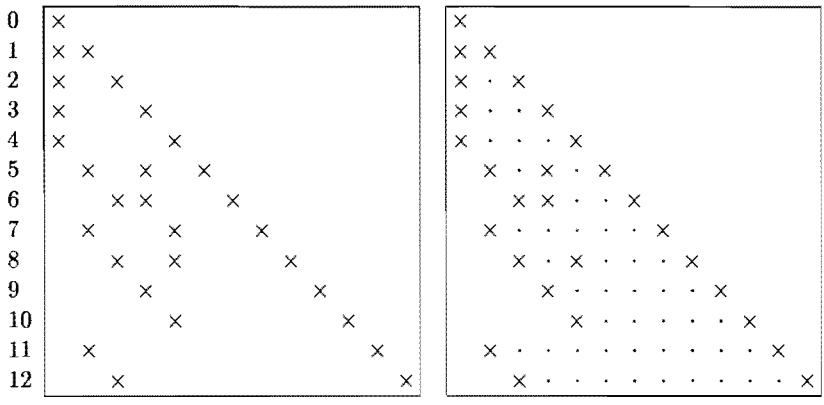


Figure 5.0: The non-zero pattern of a 13 by 13 matrix A (left) and its Cholesky factor L (right) are displayed (only the lower half of a symmetric matrix is shown). A $\times$ indicates the presence of a non-zero; a $\cdot$ the presence of a created non-zero. In the Cholesky factor 47 non-zeros are created.

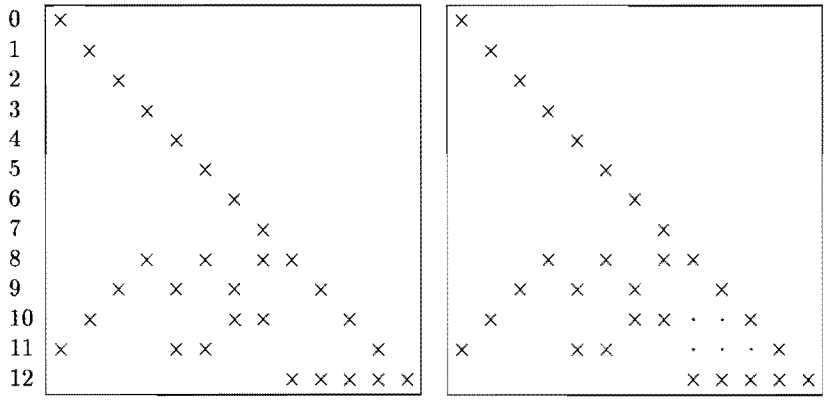


Figure 5.1: The non-zero pattern of a 13 by 13 matrix $\tilde{A}$ (left) and its Cholesky factor $\tilde{L}$ (right) are displayed. $\tilde{A}$ is obtained by symmetrically permuting the rows and columns of A (see Figure 5.0), i.e., $\tilde{A} = P\,A\,P^t$ for a suitable permutation matrix P. Observe that the *fill* is much less: only 5 non-zeros are created in the Cholesky factor $\tilde{L}$.

The problem of finding a permutation matrix P such that the number of non-zeros in L is minimal is known as the minimum-fill reordering problem, and has been proved to be NP-complete [78]. In the past decade, however, two heuristics, namely the minimum degree algorithm and nested dissection [26] have been shown to be very effective in reducing the fill. Notably, the minimum degree algorithm and its

improvements (see [30] for a review) yield, in general, a 'good' ordering for a large class of symmetric positive-definite matrices. The nested dissection ordering is less general, but it is quite effective for the class of matrices arising from discretisations of partial differential equations on rectangular grids and L-shaped domains. Pre-ordering steps like the minimum degree algorithm take only a small fraction of the time compared to the actual Cholesky factorisation.

The Cholesky factorisation can be modelled using graph theory.

Definition 5.0 (G(A)) The graph $G(A) = (V, E)$ belonging to a symmetric n by n matrix A has the set $V = \{i : 0 \leq i < n : i\}$ as vertices. The set of edges is $E = \{i, j : i, j \in V \wedge A(i, j) \neq 0 : \{i, j\}\}$. □

The following holds:

$$(\forall\, i, j : 0 \leq i, j < n \wedge \{i, j\} \notin E : A(i, j) = 0) \ .$$

The set E identifies the non-zeros of A. For convenience, we shall use the notation $A(i, j) \neq 0$ for $\{i, j\} \in E$. An example is given in Figure 5.2.

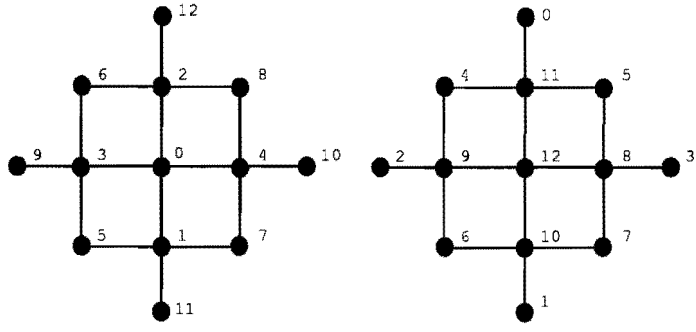


Figure 5.2: The graphs of A and $\tilde{A}$ of Figures 5.0 and 5.1 are depicted (self-loops are not drawn). They are the same except for the labeling of the vertices. The *minimum-fill reordering* problem can be restated as: find a labeling of the vertices of A such that its Cholesky factor has a minimal fill.

The sparsity pattern of the Cholesky factor L of a matrix A can be determined entirely from the graph $G(A)$. The basic equation is:

$$L(i, j) \neq 0 \equiv A(i, j) \neq 0 \vee (\exists\, k : 0 \leq k < j : L(i, k) \neq 0 \wedge L(j, k) \neq 0) \ ,$$

for all $0 \leq j \leq i < n$.

This result can be obtained using postcondition R of Chapter 4 (page 61). The equation above states that a matrix element $L(i, j)$, $i \geq j$, is a non-zero if either $A(i, j)$ is a non-zero or $L(i, j)$ is created by a pair of non-zeros $L(i, k)$ and $L(j, k)$ with $0 \leq k < j$. In terms of graph $G(A)$, $L(i, j)$ is a non-zero if there is a path in

$G(A)$ from vertex i to vertex j with all internal vertices strictly less than i **min** j, notation $i \sim j$. Note that by definition $\{i,j\} \in E$ implies $i \sim j$. A path of length t in a graph (V,E) is a sequence of vertices n_l, $0 \leq l \leq t$, such that: for all l, with $0 \leq l < t$, $\{n_l, n_{l+1}\} \in E$ holds. An internal vertex of a path $n_0 \ldots n_t$ is a vertex n_l with $0 < l < t$.

Thus, the graph $G(L) = (V, E_A)$ of the Cholesky factor L of A can be characterised by:

$$E_A = \{i,j : 0 \leq j \leq i < n \wedge i \sim j : \{i,j\}\} .$$

Example 5.1 Consider the graph of A of Figure 5.2. The path 12, 2, 8, 4, 10, has all internal vertices < 10, hence $12 \sim 10$ holds, and $L(12,10)$ is a non-zero. □

The sparsity pattern of L can be computed efficiently in $O(|E_A|)$ [29]. In the remainder of this chapter, we assume that the matrix A is reordered using a fill-reducing heuristic. Furthermore, the sparsity pattern of its Cholesky factor L is *known* a priori.

5.2 Review

In this section, a short review is given of the literature on parallel sparse Cholesky algorithms intended for distributed-memory machines. The major differences between the various approaches are discussed (see [37] for a general review on this subject).

Most sequential and parallel Cholesky factorisation algorithms are column-oriented and can be classified [37] as either column-Cholesky or submatrix-Cholesky. (Row-Cholesky is rarely considered.) The differences between both types of algorithms stem from the order in which computations are performed.

In a sequential *dense* column-Cholesky, columns are computed one by one, and each newly computed column k, $0 \leq k < n$, is modified by all previous columns j, $0 \leq j < k$. In a *sparse* column-Cholesky, each column k is only modified by columns j such that $L(k,j) \neq 0$.

The columns in a sequential submatrix-Cholesky are also computed one by one, but each newly computed column k is used to modify all columns j, $0 \leq k < j < n$. In a *sparse* version, each column k modifies only columns j such that $L(j,k) \neq 0$.

The terms left-looking algorithm and right-looking algorithm are sometimes used to distinguish column-Cholesky and submatrix-Cholesky, respectively (cf. Figure 5.3).

Almost all known parallel algorithms for the Cholesky factorisation of a sparse matrix are based on a column distribution[0] of the Cholesky factor. One of the first-published parallel implementations was a sparse submatrix-Cholesky [28] on a distributed-memory multi-processor system. This algorithm, known as the *fan-out* algorithm, uses elimination trees [68] and an arbitrary mapping of the columns to processors.

[0] A column distribution assigns entire columns to processors.

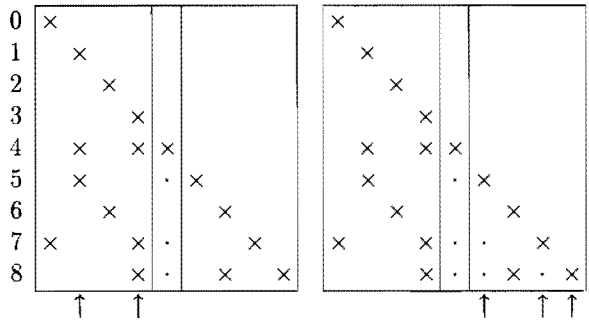


Figure 5.3: Snapshot of sequential column-Cholesky (left) and submatrix-Cholesky (right) for a 9×9 matrix. In both cases, the first 4 columns of the Cholesky factor have been computed. In the column-Cholesky, column 4 needs to be modified by the columns which are determined by the non-zeros of row 4, i.e., columns 1 and 3 as indicated by the arrows. As a consequence of this modifications, non-zeros are created in column 4. In the submatrix-Cholesky, column 4 is used to modify the columns which are determined by the non-zeros of column 4, i.e., columns 5, 7 and 8. Observe that, as a consequence of the modifications, non-zeros are created.

The fan-out algorithm and its improvements are inferior to the *fan-in* algorithm, which is a parallel column-Cholesky [0, 1]. The fan-in algorithms use either a pure column-mapping or the so-called subtree-to-subcube mapping [31]. The latter mapping uses elimination trees and works well for matrices associated with the $k \times k$ regular grid, but is difficult to generalise to more irregular problems [37].

Fan-in algorithms reduce communication overhead much better than fan-out algorithms, for example, by combining several messages into one message. This does not necessarily mean that fan-out algorithms using different data distributions, i.e., non-column distributions, are inferior too. In the following, we present a parallel submatrix-Cholesky algorithm (fan-out) based on the *grid* distribution. In Subsection 5.3.2 we demonstrate that the *grid* distribution reduces the number of communications compared to a column-based distribution. Thus, a *grid*-based submatrix-Cholesky can compete with column-based fan-in algorithms.

5.3 A parallel algorithm based on rank-1 updates

In the parallel program scheme of Chapter 4, the major source of parallelism comes from the *cmod* operations (Restore$P2$, page 65). We showed that the *grid* distribution of a dense matrix L results in an even distribution of the *cmod* operations. Moreover, the *grid* distribution avoids redistribution and consequently, the total number of communications is reduced. It is natural to use a *grid* distribution for a *sparse* matrix as well. From the previous review, it can be concluded that this is usually not done. Therefore, the *grid* distribution for a parallel sparse submatrix-Cholesky is discussed first.

5.3.0 Why *grid*?

In order to answer this question, some general observations are made. One such observation is: *The minimum degree algorithm tends to produce blocks of non-zeros in the Cholesky factor.* This is in general true, since it has been observed that groups of consecutive columns often share the same non-zero pattern ('supernodes') [63].

As discussed in Chapter 2, the two issues of distributions are load balancing and communication overhead. In a submatrix-Cholesky, work is done for some non-zeros from the active submatrix. The active submatrix in step k, $0 \leq k < n$, is defined as the square submatrix of size $n - k$ starting in diagonal element $L(k, k)$.

In a parallel submatrix-Cholesky, the work-load per process is determined by the distribution of the non-zeros in each active submatrix across the processes. The total number of communications in the algorithm is determined by the number of non-zeros a process has to send to another process during the computation.

The following observations are made.

- *The grid distribution ensures an even distribution of the rows and columns of each active submatrix.* This does not guarantee that the number of non-zeros in each active submatrix is evenly distributed across the processes, but at least structural load imbalance is avoided. This is in contrast to the *block* distribution, which causes a large load imbalance. Note that the column-*wrap* distribution also ensures an even distribution of the rows and columns of each active submatrix.

  Often, at the end of a sparse factorisation algorithm, some large dense submatrices have to be factored. In Chapter 4, it has been demonstrated that the *grid* distribution is preferred for a dense factorisation.

- *The grid distribution scatters each rectangular block of non-zeros across the processes.* This scatter property has been observed by [73]. If the sizes of blocks of non-zeros are large compared to M (M^2 is the total number of processes), then it is expected that the non-zeros in the rectangular block are distributed evenly across the processes.

- *The minimum degree algorithm tends to produce blocks of non-zeros in the Cholesky factor.* This means that the *grid* distribution is expected to achieve a good load balance with pre-ordered matrices.

- If it is necessary to replicate a *grid*-distributed column then this can be implemented efficiently. Each column is distributed across M processes, hence replicating a column to all process columns can be done by M simultaneous broadcasts.

  If the non-zeros in a column are part of a rectangular non-zero block then it is expected that the non-zeros are evenly distributed across a process column,

thus spreading the communications across the processes. Similar arguments hold for a *grid*-distributed row.

This is in contrast to a pure column-based distribution, which assigns an entire column to a process. A replication of a column requires all the non-zeros of it to be communicated from one process to all other processes. As a consequence, the communications are not spread.

In Subsection 5.3.2, we obtain an upper bound for the number of communications when using a *grid* distribution for the Cholesky factor. It is shown that the *grid* distribution reduces the number of communications with a factor of $\sqrt{p}$ compared to column distributions.

5.3.1 A parallel sparse submatrix-Cholesky algorithm

The non-zeros of L are assigned to processes according to *grid* distribution (cf. Figure 5.4):

$$\begin{aligned}(\forall\, i,j \;&:\; 0 \le j \le i < n \wedge L(i,j) \neq 0 \\ &:\; L(i,j) \text{ is assigned to process } (i \backslash M, j \backslash M))\ .\end{aligned}$$

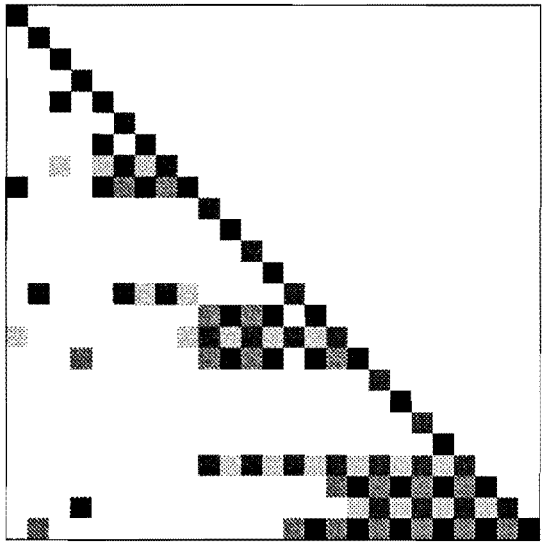


Figure 5.4: The *grid* distribution of the non-zero pattern of a 25 by 25 sparse Cholesky matrix across 4 processes is displayed. The processes are identified by different grey-shadings. This matrix is obtained from the linear programming problem 'afiro', which is in the NETLIB library [25, 3]. The matrix has been pre-ordered in order to limit fill.

The parallel program for sparse submatrix-Cholesky is obtained from the general parallel program scheme for *dense* Cholesky (see Chapter 4) by instantiating it with the *grid* distribution and exploiting sparsity.

The outline of each parameterised process (s,t) is similar as in the dense case (cf. Figure 4.1). The sparsity is exploited, for instance, in the initialisation, which sets L to A for the non-zeros of A only. Each parameterised process (s,t) consists of a loop in which Restore$P1$ and Restore$P2$ are called one after the other. An outline of the sparse version for Restore$P1$ is given in Figure 5.5.

In the annotations, predicate $sum.a.b.c$ is used (see page 61 for its definition).

```
Restore P1.s.t ::
[[ h: real;
   if  s = t ∧ k\M = t
   → {L(k,k) = sum.k.k.k}
       L(k,k) := √L(k,k)
     {L(k,k)² = sum.k.k.k}
     ; h := L(k,k)
     {h = L(k,k)}
     ; C0.s
   [] s ≠ t ∧ k\M = t → C0.s {h = L(k,k)}
   fi
 ; if  k\M = t
   → {h = L(k,k)}
       for all i : k+1 ≤ i < n ∧ i\M = s ∧ L(i,k) ≠ 0 :
           {L(i,k) = sum.i.k.k}
           L(i,k) := L(i,k)/h
           {L(i,k) * L(k,k) = sum.i.k.k}
       lla rof
   fi
]]
```

Figure 5.5: Program text for Restore$P1$ in the *grid*-based submatrix-Cholesky. k is the step counter.

The program text for a sparse version of Restore$P2$ is sketched in Figure 5.6. Arrays c and d are used to store communicated matrix elements, and they are implicitly initialised by setting all local component of the arrays to zero. The nested **for all**-statement uses the predicates $c(i) \neq 0$ and $d(j) \neq 0$. This is meaningful, since the non-zero values in each array c in process (s,t) are copies of the non-zeros $L(i,k)$ with $i\backslash M = s$. A similar remark holds for array d.

The resulting parallel Cholesky algorithm is called a single-rank update algorithm, since in the inner loop a vector addition (saxpy) is performed.

The specifications of the communication processes $C0$, $C1$ and $C2$ resemble their

```
Restore P2.s.t ::
[[ c, d(i : 0 ≤ i < n): array of real;
   if k\M = t
   → for all i : k + 1 ≤ i < n ∧ i\M = s ∧ L(i, k) ≠ 0 :
         c(i) := L(i, k)
     lla rof
   fi
  {k\M ≠ t ∨ (∀ i : k + 1 ≤ i < n ∧ i\M = s ∧ c(i) ≠ 0 : c(i) = L(i, k))}
  ; C1.s.t
  {(∀ i : k + 1 ≤ i < n ∧ i\M = s ∧ c(i) ≠ 0 : c(i) = L(i, k))}
  ; C2.s.t
  {(∀ j : k + 1 ≤ j < n ∧ j\M = t ∧ d(j) ≠ 0 : d(j) = L(j, k))}
  ; for all j : j\M = t ∧ k + 1 ≤ j < n ∧ d(j) ≠ 0 :
        for all i : i\M = s ∧ j ≤ i < n ∧ c(i) ≠ 0 :
            L(i, j) := L(i, j) − c(i) * d(j)
        lla rof
   lla rof
]]
```

Figure 5.6: Program text for Restore$P2$ in a *grid*-based submatrix-Cholesky

dense counterparts. For this reason, we omit the formal specifications and give only an informal list of the communication requirements.

- $C0.s.t$: This communication process broadcasts the value of $L(k,k)$ from process $(k\backslash M, k\backslash M)$ to each process $(s, k\backslash M)$, $0 \leq s < M$.
- $C1.s.t$: This communication process replicates the non-zero values of L's kth column across process columns. This can be done efficiently by letting each process $(s, k\backslash M)$ broadcast its non-zeros $L(i,k)$ to each process (s,t) in the same process row.
- $C2.s.t$: This communication process replicates the non-zero values of L's kth column across process rows. This can be done efficiently by letting each diagonal process (t,t) broadcast the non-zero values of c, received by $C1.t.t$, to each process (s,t) in the same process column.

The communication processes can be slightly optimised at the end of the factorisation process, but for simplicity we omit these optimisations.

5.3.2 Complexity analysis

In this subsection, a complexity analysis is given for the parallel sparse submatrix-Cholesky. The complexity results are obtained under the following assumptions.

- Every column has the same number of non-zeros c_L, $c_L \ll n$.
- Each process has c_L/M non-zeros of each column.

From these assumptions can be deduced that the non-zeros are distributed evenly over the processes. Although these assumptions are somewhat unrealistic, the results are still meaningful. Constant c_L can be looked upon as the average number of non-zeros, and an even spread is more or less guaranteed when $M \ll c_L$.

First, the complexity T_{P1} of Restore$P1$ in step k is discussed. Process $C0.s$ implements a broadcast of $L(k,k)$ which takes at most $M-1$ communications. It takes at most c_L/M divisions to complete Restore$P1$, since every column has the same number of non-zeros. This gives a total complexity:

$$T_{P1}.M.n.k = \mathbf{O}(c_L/M + \alpha * M) \ .$$

The complexity of Restore$P2$ is obtained as follows. The two broadcasts of column k by communication processes $C1.s.t$ and $C2.s.t$ each have a complexity of $c_L/M + M - 2$, since at most c_L/M elements have to be communicated to at most $M-1$ processes and pipe-lining can be used (see page 20). All communication streams can operate simultaneously. The inner **for all**-statement consists of at most c_L/M update operations $L(i,j) := L(i,j) - c(i) * d(j)$. Each process performs this inner loop at most c_L/M times, hence the complexity of Restore$P2$ in step k becomes:

$$T_{P2}.M.n.k = \mathbf{O}(\tfrac{{c_L}^2}{p} + \alpha * (c_L/M + M)) \ .$$

The total complexity of the parallel sparse submatrix-Cholesky is:

$$\begin{array}{ll} & T.p.n \\ = & \quad \{ \text{ definition } \} \\ & (\sum\, k : 0 \le k < n : T_{P1}.M.n.k + T_{P2}.M.n.k) \\ = & \quad \{ \text{ definitions } T_{P1} \text{ and } T_{P2}, \text{ calculus, } M = \sqrt{p} \ \} \\ & \mathbf{O}(n * \tfrac{{c_L}^2}{p} + \alpha * n * \tfrac{c_L}{\sqrt{p}} + \alpha * n * \sqrt{p}) \ . \end{array}$$

A number of observations can be made. If ${c_L}^2$ is of the same order of magnitude as p then the second and the third order terms become dominant. The lower order terms merely represent communication cost, which can be quite high for large values of α.

In the following, an upper bound is obtained for the total number of communications $Ncom^{grid}$ for the *grid*-based submatrix-Cholesky. Consider a column k of the Cholesky factor. Let $c_L.k$ be the number of non-zeros in column k below the diagonal. Due to the *grid* distribution, column k is distributed across process column $k \backslash M$. Suppose, furthermore, that each process $(s, k \backslash M)$ has $local.k.s$ non-zeros from column k below the diagonal. The following holds obviously:

$$c_L.k = (\sum\, s : 0 \le s < M : local.k.s) \ .$$

The number of communications associated with column k is:

$$
\begin{array}{ll}
 & (\sum\, s : 0 \le s < M : 2 * (M-1) * local.k.s) + M - 1 \\
= & \quad \{ \text{ calculus, definition } c_L.k \ \} \\
 & (M-1) * (2 * c_L.k + 1) \ .
\end{array}
$$

Explanation: diagonal element $L(k,k)$ is sent to at most $M-1$ processes; each process $(s, k\backslash M)$ broadcasts $local.k.s$ non-zeros to at most $M-1$ other processes using communication process $C1.s.t$; each process (s,s) broadcasts $local.k.s$ non-zeros to at most $M-1$ other processes using $C2.s.t$.

Hence, the total number of communications $Ncom^{grid}$ for the *grid*-based submatrix-Cholesky is:

$$
\begin{array}{ll}
 & Ncom^{grid} \\
= & \\
 & (\sum\, k : 0 \le k < n : (M-1) * (2 * c_L.k + 1)) \\
= & \quad \{ \text{ calculus, } nz_L = (\sum\, k :: c_L.k + 1) \ \} \\
 & (M-1) * (2 * nz_L - n) \ ,
\end{array}
$$

where nz_L gives the total number of non-zeros in the Cholesky factor L. Using the relation $p = M^2$ gives:

$$Ncom^{grid} = (\sqrt{p} - 1) * (2 * nz_L - n) \ .$$

Theoretical results for the complexity of sparse *parallel* algorithms are scarce. One such a result is given in [31]. There, communication results are given for matrices A of size $k^s \times k^s$ associated with regular grid of dimension $s \ge 2$. The parallel algorithms used are *fan-in* algorithms based on a *wrap* column-distribution (wrap-around task assignment) and the subtree-to-subcube mapping. For convenience, we restrict ourselves to $s = 1$.

Consider the matrix A belonging to the $k \times k$ regular grid that is pre-ordered by nested dissection. The total number of communications to factor matrix A is [31]:

$$Ncom^{wrap} = \mathbf{O}(p * k^2 * \log k)$$

for the *wrap* column-distribution, and

$$Ncom^{sub} = \Theta(p * k^2)$$

for the subtree-to-subcube mapping. Moreover, the result for the latter is asymptotically optimal, and the each process has $\Theta(k^2)$ communications to perform.

It is well known that nz_L is $\Theta(k^2 * \log k)$ [26], thus:

$$Ncom^{grid} = \mathbf{O}(\sqrt{p} * k^2 * \log k) \ .$$

The *grid* distribution reduces the total number of communications by a factor of $\sqrt{p}$ compared to the *wrap* distribution. Furthermore, the *grid* distribution has an order of magnitude fewer communications compared to the subtree-to-subcube mapping if

$$\log k_2 \leq \sqrt{p} \, .$$

Only for values of k such that $k \gg p$ the subtree-to-subcube mapping has fewer communications, but then communication time is not dominant anymore (the number of computations of the $k \times k$ regular grid is $\Theta(k^3)$ [26], hence a well-balanced parallel program has $\Theta(\frac{k^3}{p})$ computations to perform in each process).

The reduction in communication volume of the *grid* distribution has also been observed by [67] and [73].

5.3.3 Experiments

In this subsection, we present timing results of an implementation of the parallel sparse submatrix-Cholesky sketched previously. The input set of symmetric positive-definite matrices is obtained from the Harwell-Boeing library [17]. The matrices originate from different problem fields as indicated in Table 5.0. The problem sizes are modest, ranging from symmetric systems with 1072 unknowns up to 10000 unknowns.

Name	n	nz_A	nz_L	description
can1072	1072	6758	28307	airplane structure
bcspwr09	1723	4117	7252	power network
lshp1882	1882	7393	80859	L-shaped grid
lshp3466	3466	13681	183123	L-shaped grid
gr6464	4096	20098	102879	square grid
bcspwr10	5300	13571	28306	power network USA
gr100100	10000	49402	298946	square grid

Table 5.0: The test set of Harwell-Boeing matrices. The problems gr6464 and gr100100 are not in the library; they represent the $k \times k$ regular grid for $k = 64$ and $k = 100$. n gives the dimension of the matrix, nz_A and nz_L give the number of non-zeros in A and L, respectively.

The matrices are pre-ordered using a minimum degree algorithm in order to reduce the fill. From Table 5.0 it can be seen that some problems have a large fill in the Cholesky factor. For example, the matrix of problem gr100100 contains initially only 49402 non-zeros; its Cholesky factor contains 298946 non-zeros. The Cholesky factor may still be considered sparse since only 0.6 % of the total number of matrix elements are non-zero.

In Table 5.1, timing results are given for the sparse parallel submatrix-Cholesky algorithm on 4–256 transputers. The multi-processor system is an FT400-Parsytec

Name	1	4	16	64	256
can1072	15.81	5.62	2.45	1.31	0.88
bcspwr09	1.53	1.12	0.95	0.85	0.80
lshp1882	63.77	20.5	7.89	3.60	2.08
lshp3466	167.1	52.33	19.04	8.23	4.48
gr6464	55.64	19.34	8.47	4.52	3.15
bcspwr10	6.50	4.18	3.27	2.75	2.49
gr100100	212.03	-	27.43	13.57	8.75

Table 5.1: Timing results for the Harwell-Boeing problems on 1, 4, 16, 64, and 256 transputers (time in seconds).

machine consisting of 400 transputers, each having 2 Mbyte memory, arranged in a square-mesh communication network. Timing results for the $p = 1$ version are obtained on a different transputer with a 16 Mbyte memory. All computations are done in double-precision arithmetic (64 bits), and the program is coded in the parallel language Occam 2 [43]. No results for $p = 4$ on the gr100100 problem could be obtained due to memory limitations. As can be seen from the table, the execution times decrease with the number of processors. In principle, an increase by a factor of two in the number of processors can result in a similar speed-up of the computation. In practice, the gains are much less. For the problems bcspwr09 and bcspwr10 the execution times are only reduced by a small factor. The largest gains are obtained with small numbers of processors. For example, gr6464 decreases from 55.64 seconds on 1 processor to 4.52 on 64 processors. The highest speed-up is obtained for the lshp3466 problem, namely 37 on 256 processors.

The problems taken for the Harwell-Boeing library are relatively small, which results in the execution times for most problems being bounded by communication time (the $\mathbf{O}(\alpha * n * \frac{c_L}{\sqrt{p}})$ term in the complexity results). This can clearly be seen in Table 5.2 where the number of transputers is further increased to 400. Saturation in the execution times occurs, and for most problems the execution times even increase, which is caused by the third-order term $\mathbf{O}(\alpha * n * \sqrt{p})$ in the complexity formula.

Name	256	400
can1072	0.68	0.73
bcspwr09	0.25	0.27
lshp1882	1.76	1.79
lshp3466	3.89	3.68
gr6464	2.12	2.09
bcspwr10	0.80	0.82
gr100100	5.74	5.62

Table 5.2: Timing results for the Harwell-Boeing problems on 256 and 400 transputers (time in seconds).

Timing results for some larger problems originating from linear programming prob-

lmes have been given in [3]. There, it is shown that for these larger problems the execution times are considerably reduced even on 400 transputers.

In conclusion, for a modest number of processors, say up to 64, a large decrease in the execution times of the Harwell-Boeing problems is found for the *grid*-based submatrix-Cholesky. Increasing the number of processors leads to a saturation in the timing results or even an increase in time. This is mainly due to the dominance of the number of communications. It is expected that the scaling behaviour of the algorithm is much better for problems with a high number of average non-zeros per row/column (compared to $\sqrt{p}$).

As a comparison, we include in Table 5.3 timing results reported for the problem gr6363 on an Intel iPSC multi-processor system ($p = 16$) [30].

Name	*wrap*	*subtree*	$Ncom^{wrap}$	$Ncom^{sub}$
gr6363	62.34	42.17	1219769	697088

Table 5.3: Timing results taken from [30] on 16 processors of the Intel iPSC. In the columns *wrap* and *subtree*, the execution times are reported for the two parallel fan-in Cholesky programs using a *wrap*-column distribution and the subtree-to-subcube mapping, respectively. In the columns $Ncom^{wrap}$ and $Ncom^{sub}$ the number of communications are reported. The execution times can not be compared, since different architectures and programming languages are used. The number of communications, however, can be compared: for the similar problem gr6464 we find $Ncom^{grid}$ is at most 604986 (using the upper bound for $Ncom^{grid}$ on page 92).

5.4 A parallel multiple-rank update algorithm

The parallel sparse Cholesky factorisation algorithm of the previous section computes columns one by one, in order of increasing column number. The parallelism comes entirely from the distribution of data. In general, the Cholesky factorisation of a sparse matrix may use an additional source of parallelism. Many columns of the matrix L may be computed in parallel, i.e., these columns are independent, since the matrix is sparse.

This can be used to combine several single-rank updates into one multiple-rank update. Instead of computing columns one by one, a batch of columns can be computed in a single step. Communications can be also combined in large batches, thereby decreasing various communication overheads. Communication is pipe-lined; an increase in the number of values to be communicated along a pipe results in an improvement of the overall efficiency, since the startup time of the pipe becomes less important. Large batches of computations and communications also decrease the number of synchronisations, and improve the load balance.

In the following, we indicate how the use 'natural' parallelism in the parallel sparse submatrix-Cholesky. Before doing so, we discuss first elimination trees.

5.4.0 Elimination trees

The dependencies between the columns are captured by the directed *elimination graph* $T(A) = (V, E)$ associated with the Cholesky factor L of A, which is defined as follows.

Definition 5.2 (Elimination graph) The elimination graph $T(A) = (V, E_L)$ associated with the Cholesky factor L of A has the same vertex set as $G(A)$. The set of directed edges is

$$E_L = \{j, k : k = (\mathbf{min}\ i : j < i < n \wedge L(i, j) \neq 0 : i) : (j, k)\}\ .$$

By convention, $\mathbf{min}\ \emptyset = +\infty$. □

Clearly, if $(j, k) \in E_L$ and $(j, k') \in E_L$ then $k = k'$ holds. In general, graph $T(A)$ is a forest. For simplicity, we assume that the graph is a tree, the *elimination tree* [53, 68]; it has a root $n - 1$ and all its edges are directed towards the root.

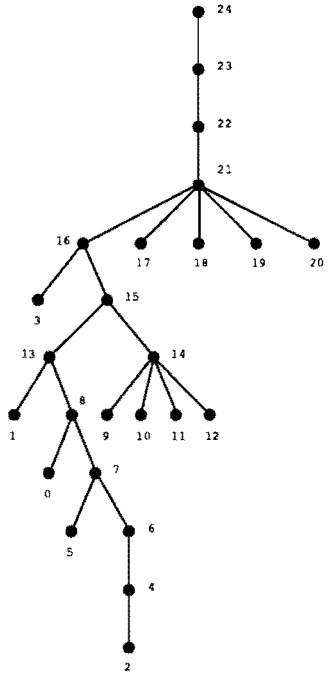


Figure 5.7: The elimination tree $T(A)$ of the NETLIB problem afiro (see Figure 5.4) is given here. The edges are depicted undirected. The interpretation of an edge (j, k), with $j < k$, in the elimination tree is that the computation of column j must precede the computation of column k. Note that $n - 1$ (=24 in this case) is the root.

The elimination tree $T(A)$ can be computed efficiently as part of the pre-ordering of A (and L), because it only depends upon the sparsity structure of L. An example of

an elimination tree is given in Figure 5.7. For a review of the use of elimination trees in Cholesky factorisation, see [54].

5.4.1 Layered-defoliation strategy

Two columns j and k, with $j < k$, are said to be independent if column j is not needed to compute column k. This is equivalent to $L(k,j)$ is a zero element. (Column k needs only the columns which are determined by the non-zeros in row k.) Independent columns can be computed in parallel, i.e., the order in which they are computed is immaterial.

Consider two vertices j and k, $j < k$, in elimination tree $T(A)$. Let $T.k$ be the set of vertices of the subtree rooted at vertex k. The following holds:

$$(*) \quad L(k,j) \neq 0 \Rightarrow j \in T.k \ .$$

This can be proved easily by induction to the 'distance' $k - j$.

If $k - j = 1$ then $(j,k) \in E_L$, hence $j \in T.k$. If $k - j > 1$ then let k' be the smallest number such that $j < k' \leq k$ and $L(k',j) \neq 0$ holds. If $k' = k$ then again $(j,k) \in E_L$, hence $j \in T.k$. In the other case, $k' < k$, $L(k',j) \neq 0$, and $k' - j < k - j$ holds, and by applying the induction hypothesis we conclude that $j \in T.k'$. Furthermore, $L(k,k') \neq 0$ must hold, and by applying the induction hypothesis again, we conclude $k' \in T.k$. Combining $j \in T.k'$ and $k' \in T.k$ gives $j \in T.k$.

The converse of $(*)$ states that every vertex j that is not a member of the subtree rooted at k is independent from k. Obviously, the leaves of the elimination tree are mutually independent, so that all the corresponding columns can be computed independently. (A generalisation is given in [54]).

This suggests the strategy of *layered defoliation* of the elimination tree: compute all columns corresponding to the leaves of the tree, remove the leaves, and repeat this until the tree is empty. The leaves that are removed in each round form a *layer* of vertices in the elimination tree. Formally:

Definition 5.3 (layered defoliation) Let H be the number of vertices on the longest directed-path in elimination tree $T(A) = (V, E_L)$. The layers obtained by the *layered defoliation* strategy are:

$$\mathcal{L}.h = \{k : k \in V \wedge \sigma.k = h : k\} \ , 0 \leq h < H,$$

where $\sigma.k$ gives the Strahler number of vertex k in a tree. □

Definition 5.4 (Strahler number) The Strahler number of a vertex k in a tree $T(A)$ is defined by:

$$\sigma.k = (\mathbf{max}\ j : (j,k) \in E_L : \sigma.j + 1) \ ,$$

and by convention $\mathbf{max}\ \emptyset = 0$, i.e., the Strahler number of a leaf is zero. □

Example 5.5 Using the *layered-defoliation* strategy, we find for the elimination tree of Figure 5.7 the following 12 ($H = 12$) layers: {0, 1, 2, 3, 5, 9, 10, 11, 12, 17, 18, 19, 20}, {4, 14}, {6}, {7}, {8}, {13}, {15}, {16}, {21}, {22}, {23} and {24}. In this example, a chain is obtained after removing the first two layers. No more parallelism is available, and each column has to be computed one by one. □

Within a layer the order of computations are immaterial; it is only required that the layers are processed starting from the leaves to the root. Hence, the columns within each layer can be renumbered consecutively. This defines a permutation π, which can be computed by the program of Figure 5.8.

```
|[ s, h: int; π(i : 0 ≤ i < n): array of int;
    s := 0 ; h := 0
  ; do h ≠ H →
        for all i : i ∈ L.h : π(i) := s + |{k : k ∈ L.h ∧ k < i : k}| lla rof
      ; s := s + |L.h|
      ; h := h + 1
    od
]|
```

Figure 5.8: Construction of a permutation vector π

Many permutations are possible that number the vertices within a layer. The chosen permutation π has the property: if $i < j$ then $\pi(i) < \pi(j)$ for any two vertices i, j from the same layer. In this way, the relative order between vertices in the same layer is maintained.

Permutation π is used to symmetrically permute the columns and rows of L (and A) such that the columns of each defoliation layer are numbered consecutively. In the following, it is assumed that A has already been permuted symmetrically using π.

Actually, the layered-defoliation strategy defines a topological ordering on the elimination tree. In a sequential algorithm, the fill and the corresponding number of operations is invariant under such an ordering [54].

The Cholesky factor L of A has a special block structure: it consists of consecutive blocks h, $0 \leq h < H$, where block h contains the $|\mathcal{L}.h|$ columns of layer $\mathcal{L}.h$. Furthermore, the $|\mathcal{L}.h| \times |\mathcal{L}.h|$ submatrix on the main diagonal of L that falls in block h is a diagonal matrix. (This can be seen as follows: the existence of a non-zero element $L(k, j)$ with $j < k$ in the submatrix implies $j \in T.k$, hence there is a path in the elimination tree from $j \in \mathcal{L}.h$ to $k \in \mathcal{L}.h$, which contradicts the fact that k and j are in the same layer $\mathcal{L}.h$.) An example of this block structure is given in Figure 5.9.

5.4.2 Using layered defoliation

Layered defoliation can be used to homogeneously distribute the independent computations, and to combine computations and communications in batches. In the fol-

lowing, we informally discuss the parallel submatrix-Cholesky based on multiple-rank updates. The matrix A has the special block structure, and is distributed according to the *grid* distribution.

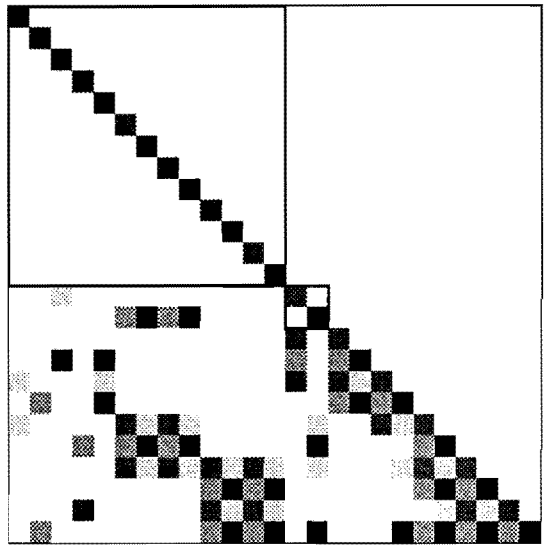


Figure 5.9: The *grid* distribution of the non-zero pattern of the permuted Cholesky factor of afiro (see Figure 5.4) across 4 processes is shown. The processes are identified by different grey-scales. The matrix is permuted according to the layered-defoliation strategy. The Cholesky factor has a special block structure: each block corresponds to one layer from the elimination tree. Only the diagonal submatrices of the first two layers are shown. The first layers consists of 13 columns, the second layer of 2 columns. The remaining layers have only one column.

The parallel multiple-rank update algorithm consists of a loop over the number of layers. In each step of the algorithm, the columns belonging to one layer are computed, and they are used to update the remaining submatrix. This is done in three phases.

In the first phase, the independent columns are computed by all processes. The *grid* distribution assigns each column to M processes, and ensures that the non-zeros in each block corresponding to a layer are distributed evenly.

In the second phase, all columns from one layer have to be replicated to all other process columns and rows, thus giving a large batch of communications.

In the third phase, each process has a number of non-zeros belonging to all the columns from a layer. These non-zeros are used to update the remaining submatrix in one large batch of computations. This results in a multiple-rank update, instead of the single-rank update in the algorithm of Subsection 5.3.1.

5.4.3 The parallel multiple-rank update algorithm

The layers are non-decreasing in size. This property can be used to compactly represent the information about the block structure of L, namely by two arrays *multi* and *rank*, each of length *nrank*. It is only necessary to record the different block sizes together with multiplicity of their occurrence: there are $multi(r)$ consecutive blocks of $rank(r)$ columns, for all r: $0 \leq r < nrank$. Clearly, the following relation holds:

$$(\sum r : 0 \leq r < nrank : multi(r) * rank(r)) = n$$

Furthermore, *rank* is strictly decreasing, and $rank(r) > 0$, $multi(r) > 0$, $0 \leq r < nrank$.

It is easy to show that:

$$n \geq (\sum r : 0 \leq r < nrank : rank(r)) \geq (nrank * (nrank - 1))/2 \ .$$

Thus, $nrank < \sqrt{2 * n}$; so arrays *rank* and *multi* are not too large.

For reasons of simplicity, we assume that $multi(r) = 1$, i.e., all blocks of matrix L have different sizes. The program text for the *grid*-based parallel submatrix-Cholesky using multiple updates is given in Figure 5.10.

```
(s,t) ::
[[ lo,r: int;
   lo := 0 ; r := 0
 ; do r ≠ nrank →
      {lo = (∑ i : 0 ≤ i < r : rank(i)) ∧ 0 ≤ r ≤ nrank}
       for all k : lo ≤ k < lo + rank(r) ∧ k\M = t :
           RestoreP1.s.t (k)
       lla rof
      {Qc.s.t}
      ; C.s.t
      {Rc.s.t}
      ; for all k : 0 ≤ k < rank(r) :
           for all j : j\M = t ∧ lo + rank(r)      ≤ j < n ∧ d(j,k) ≠ 0 :
               for all i : i\M = s ∧ j ≤ i < n ∧ c(i,k) ≠ 0 :
                   L(i,j) := L(i,j) - c(i,k) * d(j,k)
               lla rof
           lla rof
       lla rof
      ; lo := lo + rank(r) ; r := r + 1
   od
]]
```

Figure 5.10: Program text for the sparse multiple-rank update submatrix-Cholesky

The three phases of the program are clearly visible. The first phase consists of consecutive calls to Restore$P1.s.t$ (k) of Figure 5.5. For clarity, the dependence on k is expressed explicitly. The second phase consists of a large batch of communications performed by communication process $C.s.t$ with precondition $Qc.s.t$ and postcondition $Rc.s.t$. This communication process uses two-dimensional arrays c and d, each storing the necessary communicated values. In an implementation, a large buffer is sufficient to represent c and d. The specification of $C.s.t$ is not given formally, since it simply is a generalisation of $C1.s.t$ and $C2.s.t$ of the single update algorithm. Precondition $Qc.s.t$ states that locally the non-zero values of the columns k, with $lo \leq k < lo + rank(r)$ and $k \backslash M = t$ are available. Postcondition $Rc.s.t$ states that each process (s,t) has in arrays c and d the appropriate non-zeros of the columns k, $lo \leq k < lo + rank(r)$. The third phase uses the arrays c and d to update the remaining submatrix in one large batch of computations.

The resulting program is a generalisation of the single-rank update algorithm, which can be recovered if we choose $nrank = 1$, $rank(0) = 1$, and $multi(0) = n$.

A complexity analysis of the algorithm is not given here, since it is similar to the one given in Subsection 5.3.2.

5.4.4 More experiments

As in Subsection 5.3.3, the set of Harwell-Boeing matrices is taken for the timing experiments. The results for the multiple-rank update algorithm are obtained on the same hardware and software platform (cf. Table 5.4). The major difference between

Name	1	4	16	64	256
can1072	15.81	5.28	2.16	1.08	0.68
bcspwr09	1.53	0.72	0.49	0.33	0.25
lshp1882	63.77	20.0	7.20	3.19	1.76
lshp3466	167.1	52.27	18.64	7.67	3.89
gr6464	55.64	19.08	8.21	3.89	2.12
bcspwr10	6.50	2.97	1.87	1.20	0.80
gr100100	212.03	-	25.87	11.64	5.74

Table 5.4: Timing results for the Harwell-Boeing problems running the parallel multiple-rank update submatrix-Cholesky on 4, 16, 64, 256 transputers (time in seconds). The timing results for the $p = 1$ version are obtained from the rank-1 update algorithm.

Table 5.1 and this table is that the the parallel multiple-rank update algorithm is faster in all cases. The gains can be considerable, for example, bcspwr10 takes 2.49 seconds in the single-update algorithm and only 0.80 seconds in the multiple-update version. This results in higher speed-up numbers: gr100100 increases its speed-up from 14 to 37 on 256 processors. The highest speed-up is obtained for problem lshp3466, namely 43 on 256 processors. It is remarkable that the gains increase rel-

atively more with the number of processors, for example, gr100100 decreases from 13.57 to 11.64 seconds on 64 processors, and from 8.75 to 5.74 seconds on 256 processors. This is mainly caused by the fact that the improved version decreases the term $\mathbf{O}(\sqrt{p} * n)$ in the complexity results, which becomes more important with an increasing number of processors. From the timing results can be concluded that it is worthwhile to exploit 'natural' parallelism by using the layered-defoliation strategy. The additional overhead in re-ordering the matrix and the arrays *rank* and *multi* to record the block structure of L is small.

5.5 Final remarks

In this chapter, parallel algorithms for sparse Cholesky factorisation have been discussed. A parallel submatrix-Cholesky has been developed that uses the *grid* distribution of the non-zeros. It has been shown for a model problem that the *grid* distribution reduces the total number of communications by a factor of $\sqrt{p}$ compared to any column-distribution. The resulting algorithm is obtained from the parallel program scheme for *dense* Cholesky factorisation in Chapter 4. Characteristic of the parallel algorithm is that it repeatedly performs rank-1 updates with newly computed columns.

A generalisation of the rank-1 update algorithm is the multiple-rank update algorithm, which combines several columns in each step of the computation. This algorithm is obtained by exploiting 'natural' parallelism in the form of independent columns. These columns are easily discovered from the elimination tree of the Cholesky factor. A layered-defoliation strategy of this tree is used to define a renumbering that allows the independent columns to be homogeneously distributed across the processes. As a consequence of this strategy, the resulting matrix of the Cholesky factor has a special block structure. The *grid* distribution of such a matrix ensures an even distribution of independent columns, and results in a parallel algorithm that combines communications and computations in large batches. This gives a considerable reduction in the execution time of the Cholesky factorisation for problems taken from the Harwell-Boeing library.

The multiple-rank update still allows for many improvements. For example, columns often have a similar non-zero structure and can be combined to form 'supernodes'. Updates with these supernodes can be implemented more efficiently. Communications of several columns combined in a supernode can be reduced by at most a factor of two, since it is only necessary to communicate the numerical values of the different columns and the non-zero structure (the row indices) of the supernode. Another improvement would be the exploitation of the assignment of columns in the layered-defoliation strategy. Columns in the same layer are now assigned to different process columns. It is also possible to assign columns in the same layer in such a way that operations can be performed entirely local. In this way, communication can be reduced.

Chapter 6

Epilogue

6.0 Retrospect

In this thesis we advocated a design method for parallel programs. The way we design parallel programs closely resembles sequential programming. This is mainly due to the strict rules we force upon the 'structure' of a parallel program:

- A parallel program consists of p instances of a single parameterised process S.
- S is further refined by using standard sequential programming techniques into a sequence of ordinary sequential programs and communication processes, each being a parameterised process again.
- Instances of a parameterised communication process form a communication-closed layer: communication takes place only between the 'same' process instances.

As a consequence of this structure, we can consider a parallel program to be decomposed into layers. In the computation layer, work is distributed across the p processes, and each process performs computations on its set of local data. In the communication layer, the processes interact via message passing.

The decomposition into layers facilitates the correctness concerns. Each layer is constructed by using parameterised invariants, and can be proven correct by applying proof rules.

A communication layer has a separate specification usually with a simple functionality. Alternative implementations of the communication processes, which are based on different communication networks, can be analysed easily.

Preferably, communication layers should be 'thin' and avoided whenever possible. The efficiency of a parallel program is largely determined by the data distribution used, which in turn determines the work-load distribution and the number of associated communications.

The rôle of distributions has been discussed in Chapter 2. There, we discussed static distributions of arrays and matrices, and two examples of making new distributions from old ones: composition and Cartesian product. In general, reducing the total number of communications and spreading communications evenly across processes results in the formation of 'thin' communication layers.

On the other hand, computation layers should be 'thick' and well-balanced, thus reducing possible waiting times.

An example of a parallel program solving a class of segment problems was given in Chapter 3. In the derivation, the postcondition was rewritten into local and global expressions. The local expressions resulted in computation layers, and the global expressions in communication layers. Divide-and-conquer rules were obtained that combine the local and global expressions.

A larger example was given in Chapter 4, where parallel programs were obtained for dense Cholesky factorisation and triangular system solving. The Cholesky factorisation program uses Cartesian distributions, and it has been argued that the *grid* distribution has favourable properties. Timing-experiments confirm this claim. In the derivation for the triangular system solver a non-trivial communication process was obtained. Additionally, the concept of computational deadlock was discussed and the order between parameterised invariants was formalised by the use of a ranking function. Again, the resulting programs consist of a decomposition into layers.

In Chapter 5 two new parallel algorithms were obtained for the sparse Cholesky factorisation of a matrix. Both algorithms use the *grid* distribution for the non-zeros of the matrix. The first algorithm is similar to dense submatrix-Cholesky: sparsity is only exploited in a trivial sense. The second algorithm exploits independent computations, which are easily identified if the sparse matrix is pre-permuted using a layered-defoliation strategy of the elimination tree. This improved algorithm spreads the independent computations evenly across process columns and combines computations and communications in batches. Timing-experiments on 1–256 transputers have been given as well. The Harwell-Boeing matrices used in the experiments represent realistic problems. Although the programs have not been formally derived, the structure of the parallel programs reflects a decomposition into layers.

Our primary contribution has been to demonstrate that parallel programs can be constructed in a way which does not much differ from sequential programming.

6.1 Applications and future work

Many of the ideas found in this thesis have been applied, formally or informally, to the construction of KSLA's parallel linear algebra library. An example of a large parallel program (6500 lines of code) using this library is a parallel linear programming solver [3]. In [71] a parallel implementation of a direct fluid-flow simulator is reported, which uses a decomposition into layers.

Other examples of parallel program derivations based on parameterised invariants include dynamic programming [47] and sparse LU decomposition [72].

In our opinion, efficient parallel algorithms can be made with our method. Here, we restricted the presentation to only a few examples, but the limits of our method have been explored to some extent. For example, the number of processes can be increased until a so-called fine-grained parallel program is obtained. Of course, our method is applicable to fine-grained programs but it is our experience that the borders between the different kinds of layers then become fuzzier. Consequently, it is more difficult to give a work-load analysis. Implementations of parallel programs constructed with our method are targeted at powerful, multi-processor systems consisting of a modest number of processors — say 2–1024.

In the future we are planning to enlarge the number of applications of our method. The problem fields we will focus on are sparse matrix computations and graph algorithms. These are interesting problems because of their more irregular communication requirements and the challenges that lie in parallelising these problems.

Bibliography

[0] C. Ashcraft, S.C. Eisenstat, and J.W.H. Liu. A fan-in algorithm for distributed sparse numerical factorization. *SIAM J. Sci. Stat. Comput.*, 11(3):593–99, 1990.

[1] C. Ashcraft, S.C. Eisenstat, J.W.H. Liu, B.W. Peyton, and A.H. Sherman. A compute-ahead implementation of the fan-in sparse distributed factorization scheme. *Oak Ridge National Laboratory report ORNL/TM-11496*, August, 1990.

[2] K. van Berkel, J. Kessels, 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.

[3] R. H. Bisseling, T.M. Doup, and L.D.J.C. Loyens. A parallel interior point algorithm for linear programming on a network of transputers. *Annals of Operations Research*, 1992 (in press).

[4] R. H. Bisseling and L.D.J.C. Loyens. Towards peak parallel LINPACK performance on 400 transputers. *Supecomputer*, 8(5):20–27, 1991.

[5] R.H. Bisseling and J.G.G. van de Vorst. Parallel LU decomposition on a transputer network. In *LNCS, Parallel Computing 1988*, number 384, pages 61–77, 1989.

[6] R.H. Bisseling and J.G.G. van de Vorst. Parallel triangular system solving on a mesh network of transputers. *SIAM J. Sci. Stat. Comput.*, 12(4):787–99, 1991.

[7] H.L. Bodlaender. *Distributed Computing, Structure and Complexity.* PhD thesis, Utrecht University, 1986.

[8] K.M. Chandy and J. Misra. Distributed computation on graphs: Shortest path algorithms. *Communications of the ACM*, 25(11):833–37, 1982.

[9] K.M. Chandy and J. Misra. Systolic algorithms as programs. *Distributed Computing*, (1):177–83, 1986.

[10] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[11] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 1990.

[12] E. Dekel and S. Sahni. Binary trees and parallel scheduling algorithms. *IEEE Transactions on Computers*, (C32):307–15, 1983.

[13] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1976.

[14] E.W. Dijkstra and W.H.J. Feijen. *Een methode van programmeren.* Academic Service, Den Haag, 1984.

[15] E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics.* Springer-Verlag, New York, 1990.

[16] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices.* Oxford University Press, 1986.

[17] I.S. Duff, R.G. Grimes, and J.G. Lewis. Sparse matrix test problems. *ACM TOMS*, 15(1):1–14, 1989.

[18] T.H. Dunigan. Performance of the Intel iPSC/860 and Ncube 6400 hypercubes. *Parallel Computing*, (17):1285–1302, 1991.

[19] S.C. Eisenstat, M.T. Heath, C.S. Henkel, and C.H. Romine. Modified cyclic algorithms for solving triangular systems on distributed-memory multiprocessors. *SIAM J. Sci. Stat. Comput.*, 9(3):589–600, 1988.

[20] T. Elrad and N. Francez. Decomposition of distributed programs into communication-closed layers. *Science of Computer Programming*, (2):155–73, 1982.

[21] G.C. Fox, M.A. Johnson, G.A. Lyzenga, S.W. Otto, J.K. Salmon, and D.W. Walker. *Solving Problems On Concurrent Processors, Volume 1.* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[22] R.G. Gallager, P.A. Humblet, and P.M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems*, 5(1):66–77, 1983.

[23] K.A. Gallivan, R.J. Plemmons, and A.H. Sameh. Parallel algorithms for dense linear algebra computations. *SIAM Review*, 32(1):54–135, 1990.

[24] N. van Gasteren. Private communication. 1989.

[25] D.M. Gay. Electronic mail distribution of linear programming test problems. *Math. Program. Soc. COAL Newsl.*, (13):10–2, 1985.

[26] A. George. Nested dissection of a regular finite element mesh. *SIAM J. Numer. Anal.*, 10(2):345–63, 1973.

[27] A. George, M.T. Heath, and J.W.H. Liu. Parallel Cholesky factorization on a shared-memory multiprocessor. *Lin. Alg. Appl.*, (77):165–87, 1986.

[28] A. George, M.T. Heath, J.W.H. Liu, and E. Ng. Sparse Cholesky factorization on a local-memory multiprocessor. *SIAM J. Sci. Stat. Comput.*, 9(2):327–40, 1988.

[29] A. George and J.W.H. Liu. *Computer Solutions of Large Sparse Positive Definite Systems.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[30] A. George and J.W.H. Liu. The evolution of the minimum degree ordering algorithm. *SIAM Review*, 31(1):1–19, 1989.

[31] A. George, J.W.H. Liu, and E. Ng. Communication results for parallel sparse Cholesky factorization on a hypercube. *Parallel Computing*, (10):287–98, 1989.

[32] G.H. Golub and C.F. Van Loan. *Matrix Computations (2nd edition).* The John Hopkins University Press, 1989.

[33] D. Gries. *The Science of Programming.* Springer-Verlag New York, 1981.

[34] D. Gries. A note on a standard strategy for developing loop invariants and loops. *Science of Computer Programming*, (2):207–14, 1982.

[35] J.L. Gustafson, G.R. Montry, and R.E. Benner. Development of parallel methods for a 1024-processor hypercube. *SIAM J. Sci. Stat. Comput.*, 9(4):609–38, 1988.

[36] F. Harary. *Graph Theory.* Addison-Wesley, 1972.

[37] M.T. Heath, E. Ng, and B.W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33(3):420–60, 1991.

[38] M.T. Heath and C.H. Romine. Parallel solution of triangular systems on distributed-memory multiprocessors. *SIAM J. Sci. Stat. Comput.*, 9(3):558–88, 1988.

[39] P.A.J. Hilbers. *Mappings of Algorithms on Processor Networks.* PhD thesis, Groningen University, also as: Processor Networks and Aspects of the Mapping Problem in Cambridge International Series on Parallel Computations 2, 1989.

[40] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–80, 1969.

[41] C.A.R. Hoare. Parallel programming: an axiomatic approach. *Computer Languages*, 1(2):151–60, 1975.

[42] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, UK, Ltd., London, 1985.

[43] Inmos Ltd. *Occam 2 Reference Manual.* Prentice-Hall International, UK, Ltd., London, 1988.

[44] S.L. Johnsson. Communication efficient basic linear algebra computations on hypercube architectures. *J. Parallel Distrib. Comput.*, (4):133–72, 1987.

[45] A. Kaldewaij. *Programming: The Derivation of Algorithms.* Prentice-Hall International, UK, Ltd., London, 1990.

[46] A. Kaldewaij. Shortest and longest segments. In *Beauty Is Our Business*, pages 226–32. Springer-Verlag New York, 1990.

[47] J.M.F.M. van Kemenade. Parallel dynamic programming on a fixed processor network. *Master's Thesis, Eindhoven University of Technology*, 1989.

[48] H.J. Kim and J.G. Lee. Partial sum problem mapping into a hypercube. *Information Processing Letters*, (36):221–24, 1990.

[49] G.A.P. Kindervater and J.K. Lenstra. An introduction to parallelism in combinatorial optimization. *Discrete Appl. Math.*, (14):135–56, 1986.

[50] R.E. Ladner and M.J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27(4):831–38, 1980.

[51] G.M. Levin and D. Gries. A proof technique for Communicating Sequential Processes. *Acta Informatica*, (15):281–302, 1981.

[52] G. Li and T.F. Coleman. A parallel triangular solver for a distributed-memory multiprocessor. *SIAM J. Sci. Stat. Comput.*, 9(3):485–502, 1988.

[53] J.W.H. Liu. A compact row storage scheme for Cholesky factors using elimination trees. ACM TOMS, 12:127–48, 1986.

[54] J.W.H. Liu. The role of elimination trees in sparse factorization. *SIAM J. Matrix. Anal. Appl.*, 11(1):134–72, 1990.

[55] L.D.J.C. Loyens and R.H. Bisseling. The formal construction of a parallel triangular system solver. In *LNCS, Mathematics of Program Construction*, number 375, pages 325–34, 1989.

[56] L.D.J.C. Loyens and J.G.G. van de Vorst. Two small parallel programming exercises. *Science of Computer Programming*, (15):159–69, 1990.

[57] J.J. Lukkien. Transputer Pascal, a user manual. *Technical Report CS8912*, Groningen University, 1989.

[58] J.J. Lukkien. *Parallel Program Design and Generalized Weakest Preconditions.* PhD thesis, Groningen University, 1991.

[59] A.R. Mackintosh. Dr. Atanasoff's computer. *Scientific American*, (August):72–78, 1988.

[60] A.J. Martin. A distributed implementation method for parallel programming. In *Proc. IFIP congress 80*, pages 309–14, 1980.

[61] A.J. Martin. The probe: an addition to communication primitives. *Information Processing Letters*, (20):125–30, 1985.

[62] C.R. Mollenhoff. *Atanasoff: Forgotten Father of the Computer.* Iowa State University Press, Ames, 1988.

[63] E. Ng and B.W. Peyton. A supernodal Cholesky factorization algorithm for shared-memory multiprocessors. *Oak Ridge National Laboratory report ORNL/TM-11814*, April, 1991.

[64] S. Owicki. *Axiomatic Proof Techniques for Parallel Programs.* PhD thesis, Cornell University, 1975.

[65] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, (6):319–40, 1976.

[66] M. Rem. Small programming exercises 20. *Science of Computer Programming*, (10):99–105, 1988.

[67] P. Sadayappan and S.K. Rao. Communication reduction for distributed sparse matrix factorization on a processor mesh. In *Proc. Supercomputing '89, ACM Press, New York*, pages 135–55, 1989.

[68] R. Schreiber. A new implementation of sparse Gaussian elimination. *ACM Trans. Math. Software*, (8):256–76, 1982.

[69] J.L.A. van de Snepscheut. *Trace Theory and VLSI Design.* PhD thesis, Eindhoven University of Technology, also appeared as *LNCS* 200, 1983.

[70] J.L.A. van de Snepscheut. A derivation of a distributed implementation of Warshall's algorithm. *Science of Computer Programming*, (7):55–60, 1986.

[71] J.A. Somers and P.C. Rem. A parallel cellular automata implementation on a transputer network for the simulation of small scale fluid flow experiments. In *LNCS, Parallel Computing 1988*, number 384, pages 116–126, 1989.

[72] A.F. van der Stappen. Distributed data structures for sparse linear algebra. *Master's Thesis, Eindhoven University of Technology*, 1988.

[73] A.F. van der Stappen, R.H. Bisseling, and J.G.G. van de Vorst. Parallel sparse LU decomposition on a mesh network of transputers. *SIAM J. Matrix. Anal. Appl.*, (14), 1993 (in press).

[74] F.A. Stomp. *Design and Verification of Distributed Network Algorithms: Foundations and Applications.* PhD thesis, Eindhoven University of Technology, 1989.

[75] J.G.G. van de Vorst. The formal development of a parallel program performing LU-decomposition. *Acta Informatica*, (26):1–17, 1988.

[76] J.H. Wilkinson and C. Reinsch. *Linear Algebra.* Springer-Verlag Berlin, 1971.

[77] L.D. Wittie. Communication structures for large networks of microcomputers. *IEEE Transactions on Computers*, C-30(4):264–73, 1981.

[78] M. Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM J. Alg. Disc. Meth.*, 2(1):77–79, 1981.

Index

Samenvatting

De tijdsduur van een computerberekening kan worden bekort door gebruik te maken van een parallel computersysteem bestaande uit een vast aantal identieke processoren elk met een eigen lokaal geheugen. Elke processor kan gelijktijdig een onafhankelijk berekeningsdeel uitvoeren. Vaak is het noodzakelijk om deelberekeningen te combineren, daarom worden er tussen de processoren berichten verstuurd via verbindingskanalen. De zojuist geschetste parallelle computer beschikt dus over een groot verkaveld geheugen dat toegankelijk is door een communicatienetwerk.

In dit proefschrift wordt een methode besproken om op gestructureerde wijze efficiënte programma's te maken voor een parallelle computer met verkaveld geheugen. In het eerste hoofdstuk van het proefschrift wordt ingegaan op allerlei aspecten van de methode. Kort samengevat: een parallel programma bestaat uit p aanroepen van een enkel geparametriseerd programma. Zo'n programma is verder verfijnd in een concatenatie van gewone sequentiële programma's en communicerende processen, d.w.z. programmadelen die berichten versturen. De sequentiële programma's worden op formele wijze verkregen m.b.v. de invariantenmethode. De communicerende processen bevatten communicatieacties en zijn verantwoordelijk voor de berekening en verspreiding van globale informatie.

We eisen van een parallel programma dat het opgebouwd is uit afwisselende reken- en communicatielagen. In een rekenlaag wordt door alle sequentiële programma's een onafhankelijke berekening uitgevoerd. In een communicatielaag vindt interactie plaats tussen de verschillende communicatieprocessen van het parallelle programma. De communicatieacties behorende bij een communicatielaag zijn gesloten, d.w.z. geen enkele communicatieactie geschiedt tussen twee verschillende communicatielagen. De communicatieprocessen van een laag kunnen afzonderlijk gespecificeerd worden, waardoor het mogelijk is om alternatieve verwezenlijkingen van deze programmadelen te bestuderen.

Een efficiënt parallel programma wordt verkregen door de beoogde berekening evenwichtig over de sequentiële programma's te verdelen èn het aantal communicatielagen zo klein mogelijk te houden. De verdeling van de berekening wordt grotendeels bepaald door de verdeling van de variabelen over het verkavelde geheugen. Immers, voor de eindwaarde van elke variabele dient men een aantal bewerkingen uit te voeren. In het tweede hoofdstuk worden enige eenvoudige verdelingsfuncties voor rijen en matrices besproken.

In het derde hoofdstuk passen we de gepropageerde methode toe op een klasse van berekeningen met rijen. De verkregen parallelle programma's vertonen duidelijk een lagenstructuur; daardoor is het mogelijk om theoretische uitspraken te doen over de doeltreffendheid van de programma's.

In het vierde hoofdstuk worden parallelle programma's behandeld die de oplossing berekenen van een symmetrisch positief-definiet systeem. De matrix van zo'n systeem is dicht en de coëfficiënten worden verdeeld m.b.v. een Cartesische distributiefunctie. Vervolgens wordt het systeem opgelost door een ontbinding à la Cholesky, gevolgd door het oplossen van twee driehoeksstelsels. Er wordt een uitvoerige analyse gegeven van de werklastverdeling en de communicatieverplichtingen van het parallelle Cholesky factorisatie algoritme. De *grid* distributiefunctie, ook wel splinterafbeelding genoemd, heeft verreweg de beste eigenschappen, wat ook aangetoond is door een vergelijking met parallelle programma's die andere distributiefuncties gebruiken. De driehoeksoplosser wordt, gegeven de splinterafbeelding van de matrix, op eenvoudige wijze geconstrueerd volgens de spelregels van de programmeermethode.

In het vijfde hoofdstuk wordt gekeken naar ijle symmetrische systemen, die zich kenmerken door de aanwezigheid van grote aantallen nulcoëfficiënten. Twee parallelle Cholesky factorisatiealgoritmen worden behandeld, die elk de splinterafbeelding gebruiken. De eerste algoritme is een rechtstreekse parallellisatie van een sequentieel submatrixalgoritme. De tweede algoritme is een verbetering die gebruikt maakt van onafhankelijke pivotelementen en van een snoeistrategie van de eliminatieboom. De executietijden van beide algoritmen worden met elkaar vergeleken; daartoe worden er tijdswaarnemingen gedaan op problemen die afkomstig zijn uit de Harwell-Boeing collectie.

De programmeermethode, zoals deze hier verdedigd wordt, is toegepast op voorbeelden uit de praktijk. Gebleken is dat op deze wijze efficiënte parallelle programma's verkregen kunnen worden.

Curriculum vitae

De schrijver van dit proefschrift werd geboren op 3 december 1963 te Maastricht. Op 4 juni 1982 slaagde hij met lof voor het examen Atheneum β aan het Stedelijk Lyceum en Havo te Maastricht. Daarna werd op 6 september 1982 een aanvang gemaakt met de studie Informatica aan de indertijd geheten Technisch Hogeschool Eindhoven.

Na de voltooiing van een korte stage in de Discrete Wiskunde bij dr. ir. H.C.A. van Tilburg begon op 1 september 1986 het afstudeerwerk "An Occam machine offering full communication" onder begeleiding van prof. dr. M. Rem en ir. J.G.G. van de Vorst. Dit werk werd uitgevoerd bij het Koninklijke/Shell-Laboratorium, Amsterdam (KSLA) en resulteerde in een geparametriseerde 'pre-processor' die automatisch routeringsprocessen toevoegde aan een parallel programma.

Op 14 mei 1987 werd het doctoraal examen in de studierichting der Informatica gehaald aan de Technische Universiteit Eindhoven. Daags erna, om 8.30 's-ochtends, volgde een aanstelling bij het KSLA als software research engineer.

Sedertdien werden er door de schrijver van dit proefschrift werkzaamheden verricht in het Parallel Computing project van de afdeling Mathematics and System Engineering. Onder begeleiding van prof. dr. M. Rem werd, sinds 1988, tevens onderzoek gedaan naar het gebruik van formele methoden voor de constructie van parallelle programma's. Dit resulteerde uiteindelijk in de totstandkoming van dit proefschrift.

Stellingen
bijbehorende bij het proefschrift

A Design Method For Parallel Programs

van

L.D.J.C. Loyens

0. Voor belangrijke lineaire algebra operaties zoals matrixvermenigvuldiging, LU-decompositie, Cholesky factorisatie, QR-decompositie, en driehoekstelseloplossen bestaan er efficiënte parallelle algoritmen die gebruik maken van de *grid* distributie [10, 4, 2, 7, 5].

1. In een Cartesische productgraaf $G \times H$ geldt dat de gemiddelde padlengte gelijk is aan de som van de gemiddelde padlengten in G en H respectievelijk. Met dit gegeven kan op inzichtelijke en eenvoudige wijze, in tegenstelling tot [11], de gemiddelde padlengte in een binaire hyperkubus worden bepaald.

2. De niet-Cartesische distributiefunctie (zie [1] voor def. distributiefunctie)

$$(\mathbf{n}^2, \mathbf{M}^2, (\lambda i, j \cdot ((i + j/M)\backslash M, (j + i/M)\backslash M)))$$

van een n bij n matrix heeft de eigenschap dat elke rij en kolom van de matrix evenredig wordt verdeeld over alle M^2 processen. Met deze distributie kan een parallel matrix-vector vermenigvuldigingsprogramma worden ontworpen dat een betere werklastverdeling heeft dan met een vierkante Cartesische distributie.

3. Zij $(V, <)$ een eindig transitief systeem. Dan geldt (zie [2] voor def. ranking function):

$$(V, <) \text{ is irreflexief } \equiv (\exists r :: r \text{ is een ranking function op } (V, <)) \ .$$

4. Zij A de matrix behorende bij het regulier vierkant rooster met in iedere richting $2^k - 1$, $k \geq 1$, roosterpunten die genummerd zijn volgens 'nested dissection' [6]. De snoeistrategie van de eliminatieboom van A kan op zeer compacte wijze beschreven worden door middel van de rijen *multi* en *rank* elk van de lengte *nrank* [3]. Er geldt:

$$\begin{aligned} &nrank && = 2 * k - 1, \\ &rank(nrank - 1 - i) && = 2^i, \\ &multi(nrank - 1 - i) && = 2^{k-(i+1)/2} - 1, \qquad 0 \leq i < nrank. \end{aligned}$$

5. De **parseq** regel [0] vermijdt het gebruik van z.g. parallelle 'debuggers'. Bovendien is de $p = 1$ versie van een –met deze regel verkregen– parallel programma geschikt om te worden uitgevoerd op een sequentiële computer.

6. De parallelle programmeertaal Occam [8] leent zich uitstekend tot het maken van lange programma's, maar is ongeschikt voor grote programma's.

7. In de huidige ontwikkeling van processoren voor parallelle computersystemen wordt teveel nadruk gelegd op de Megafloppen per seconde en neemt de verhouding tussen communicatie- en rekentijd toe. Dit achten wij, met oog op de programmeerbaarheid, een ongewenste ontwikkeling.

8. In de Griekse mythologie volbrengt Heracles de twaalf werken, waaronder het verslaan van de negenkoppige Hydra. Volgens [9] is, in het gevecht met de Hydra, elke strategie een winnende, m.a.w. Heracles kon niet verliezen! Hieruit concluderen wij dat er hooguit elf echte werken waren.

9. In een organisatiestructuur zou het beoordelen van minderen door meerderen ook omgekeerd moeten plaatsvinden. Zo wordt van alle beoordeelden de wijze van functioneren beter vastgesteld.

10. Waar gehackt wordt vallen spaanders.

Referenties

[0] Hoofdstuk 1 van dit proefschrift.

[1] Hoofdstuk 2 van dit proefschrift.

[2] Hoofdstuk 4 van dit proefschrift.

[3] Hoofdstuk 5 van dit proefschrift.

[4] R.H. Bisseling and J.G.G. van de Vorst. Parallel LU decomposition on a transputer network. In *LNCS, Parallel Computing 1988*, number 384, pages 61–78, 1989.

[5] R.H. Bisseling and J.G.G. van de Vorst. Parallel triangular system solving on a mesh network of transputers. *SIAM J. Sci. Stat. Comp.*, 12(4):787–99, 1991.

[6] A. George. Nested dissection of a regular finite element mesh. *SIAM J. Numer. Anal.*, 10(2):345–63, 1973.

[7] B. Hendrickson. Parallel QR factorization on a hypercube using the torus wrap mapping. *Sandia National Laboratories, Tech. Rep. SAND91-0874.*, 1991.

[8] Inmos Ltd. *Occam 2 Reference Manual.* Prentice-Hall International, UK, Ltd., London, 1988.

[9] L. Kirby and J. Paris. Accessible independence results for Peano arithmetic. *Bulletin London Mathematical Society*, (14):285–93, 1982.

[10] L.D.J.C. Loyens. Parallel programming techniques for linear algebra. In *LNCS, Parallel Computing 1988*, number 384, pages 32–43, 1989.

[11] L.D. Wittie. Communication structures for large networks of microcomputers. *IEEE Transactions on Computers*, C-30(4):264–73, 1981.