Evaluating Parallel Algorithms: Theoretical and Practical Aspects

Lasse Natvig
Division of Computer Systems and Telematics
The Norwegian Institute of Technology
The University of Trondheim
NORWAY

November 29, 1996

Abstract

The motivation for the work reported in this thesis has been to lessen the gap between theory and practice within the field of parallel computing.

When looking for new and faster parallel algorithms for use in *massively parallel systems*, it is tempting to investigate promising alternatives from the large body of research done on parallel algorithms within the field of theoretical computer science. These algorithms are mainly described for the PRAM (Parallel Random Access Machine) model of computation.

This thesis proposes a method for *evaluating the practical value of PRAM algorithms*. The approach is based on implementing PRAM algorithms for execution on a CREW (Concurrent Read Exclusive Write) PRAM simulator. Measuring and analysis of *implemented* algorithms on *finite* problems provide new and more practically oriented results than those traditionally obtained by asymptotical analysis (O-notation).

The evaluation method is demonstrated by investigating the practical value of a new and important parallel sorting algorithm from theoretical computer science—known as *Cole's Parallel Merge Sort algorithm.* Cole's algorithm is compared with the well known Batcher's bitonic sorting algorithm. Cole's algorithm is asymptotically probably the fastest among all known sorting algorithms, and also cost optimal. Its $O(\log n)$ time consumption implies that it is faster than bitonic sorting which is $O(\log^2 n)$ time—provided that the number of items to be sorted, n, is large enough. However, it is found that bitonic sorting is faster as long as n is less than 10^{21} (*i.e.* about 1 Giga Tera items)! Consequently, Cole's logarithmic time algorithm is *not fast in practice.*

The thesis also gives an overview of complexity theory for sequential and parallel computations, and describes a promising alternative for parallel programming called *synchronous MIMD programming.*

Preface

This is a thesis submitted to the Norwegian Institute of Technology (NTH) for the doctoral degree "doktor ingeniør" (dr.ing.). The reported work has been carried out at the Division of Computer Systems and Telematics, The Norwegian Institute of Technology, The University of Trondheim, Norway. The work has been supervised by Professor Arne Halaas.

Thesis Overview and Scope

About the contents

Chapter 1 starts by explaining how the likely proliferation of computers with a large number of processors makes it increasingly important to know the research done on parallel algorithms within theoretical computer science. It outlines how the work described in the thesis was motivated by a large gap between theory and practice within parallel computing, and it presents some aspects of this gap. The chapter also summarises the main contributions of my work.

Chapter 2 describes various concepts which are central when studying, teaching or evaluating parallel algorithms. It provides an introduction to parallel complexity theory—including themes such as a class of problems that may be solved "very fast" in parallel, and another class consisting of inherently serial problems. The chapter ends with a discussion of Amdahl's law and how the size of a problem is essential for the speedup that may be achieved by using parallel processing.

Chapter 3 is devoted to the so-called CREW PRAM (Concurrent Read Exclusive Write Parallel Random Access Machine) model. This is a model for parallel computations which is well known within theoretical computer science. It is the main underlying concept for this work. After a general description of the model, it is explained how the model may be programmed in a high level notation. The programming style which has been used, called synchronous MIMD programming, contains some properties which make programming a "surprisingly easy" task. These nice features are outlined. At the end of the chapter it is given a more technical description of how algorithms may be implemented on the CREW PRAM simulator prototype—which has been developed as part of the work.

Chapter 4 describes the technically most difficult part of the work—a detailed investigation of a well-known sorting algorithm from theoretical computer science (Cole's parallel merge sort). A top-down, detailed understanding of the algorithm is presented together with a description of how it has been implemented. The implementation is probably the first complete implementation of this algorithm. A comparison of the implementation with several simpler sorting algorithms based on measurements and analytical models being presented.

Chapter 5 summarises what has been learned from doing this work, and gives a brief sketch of interesting directions for future research.

Appendix A gives a brief description of the CREW PRAM simulator prototype which makes it possible to implement, test and measure CREW PRAM algorithms. The main part of the chapter is a user's guide for how to develop and test parallel programs on the simulator system as it is currently used at the Norwegian Institute of Technology.

Appendix B gives a complete listing of the implementation of Cole's parallel merge sort algorithm. In addition to being the fundamental "documentation" of the implementation, it is provided to give the reader the possibility of studying the details of medium-scale synchronous MIMD programming on the simulator system.

Issues given less attention

The thesis covers complexity theory, detailed studies of a complex algorithm, and implementation of a simulator system. This variety has made it important to restrict the work. I have tried to separate the goals and the tools used to achieve these goals. It has been crucial to restrict the efforts put into the design of the CREW PRAM simulator system. Prototyping and simple ad hoc solutions have been used when possible without reducing the quality of the simulation results.

The pseudo code notation used to express parallel algorithms is not meant to be a new and better notation for this purpose. It is intended only to be a common sense modernisation of the notation used in [Wyl79] for high level programming of the *original* CREW PRAM model.

Similarly, the “language” used to implement the algorithms as running programs on the simulator is not a proposal of a new, complete and better language for parallel programming, just a minor extension of the language used to implement the simulator.

Simplicity has been given higher priority than efficient execution of parallel programs in the development of the CREW PRAM simulator.

Even though nearly all the algorithms discussed are sorting algorithms, the work is not primarily about parallel sorting. It is not an attempt to find new and better parallel sorting algorithms, nor is it an attempt to survey the large number of parallel sorting algorithms. Sorting has been selected because it is an important problem with many well known parallel algorithms.

Acknowledgements

I wish to thank my supervisor Professor Arne Halaas for introducing me to the field of complexity theory, for constructive criticisms, for continuing encouragement, and for letting me work on own ideas.

The Division of Computer Systems and Telematics consists of a lot of nice and helpful people. Håvard Eidnes, Anund Lie, Jan Grønsberg and Tore E. Aarseth have answered technical question, provided excellent computer facilities, and allowed me to consume an awful lot of CPU seconds. Gunnar Borthne, Marianne Hagaseth, Torstein Heggebø, Roger Midtstraum, Petter Moe, Eirik Knutsen, Øystein Nytrø, Tore Sæter, Øystein Torbjørnsen, and Håvard Tveite have all read various drafts of the thesis, and provided valuable comments. Thanks.

The work has been financially supported by a scholarship from The Royal Norwegian Council for Scientific and Industrial Research (NTNF).

Last, but not least, I am grateful to my family. My parents for teaching me to believe in my own work, my wife Marit for doing most of the research in a much more important project: taking care of our children Ola and Simen, and to Ola and Simen which have been a continuing inspiration for finishing this thesis.

Trondheim, December 1990.

Lasse Natvig

Contents

Chapter 1

Introduction

"There are two main communities of parallel algorithm designers: the theoreticians and the practitioners. Theoreticians have been developing algorithms with narrow theory and little practical importance; in contrast, practitioners have been developing algorithms with little theory and narrow practical importance."

Clyde P. Kruskal in *Efficient Parallel Algorithms: Theory and Practice* [Kru89].

1.1 Massively Parallel Computation

There is no longer any doubt that parallel processing will be used extensively in the future to build the at any time, most powerful general purpose computers. Parallelism has been used for many years in many different ways to increase the speed of computations. Bit-parallel arithmetic (the 1950'ies), multiple functional units (1960'ies), pipelined functional units (1970'ies) and multiprocessing (1980'ies) are all important techniques in this context [Qui87].

Today, these and many other ways of parallelism are being exploited by a vast number of different computer architectures—in commercial products and research prototypes. One can not predict in detail which computer architectures will be the dominating for use in the most powerful (parallel) general purpose computers in the future. In the 1980'ies the supercomputer market has been dominated by the pipelined computers (also called vector computers) such as those manufactured by Cray Research Inc., USA, and

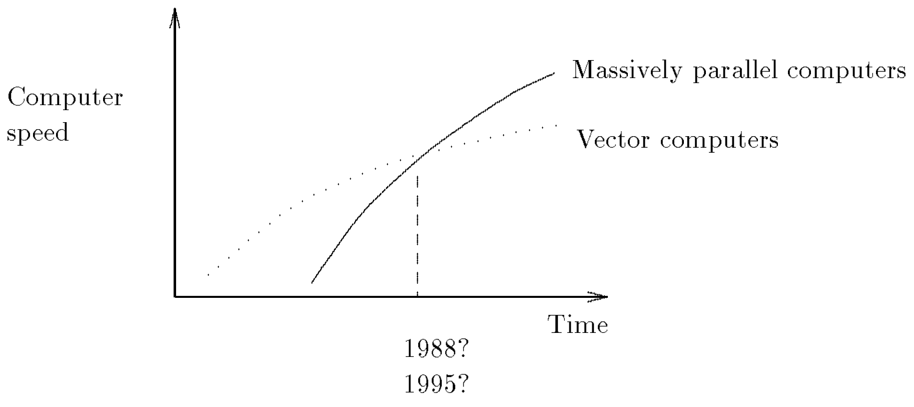


Figure 1.1: Massively parallel computers are expected to be faster than vector computers in the near future.

similar computers from the Japanese companies Fujitsu, Hitachi and Nec [HJ88]. However, the next ten years may change the picture.

At the SUPERCOMPUTING'90 conference in NewYork, November 1990, it seemed to be a widespread opinion among supercomputer users, computer scientists and computational scientists that *massively parallel computers*[1] will outperform the pipelined computers and become the dominating architectural main principle in year 2000 or earlier. The main argument for this belief is that the computing power (speed) of massively parallel computers has grown faster, and is expected to continue to grow faster than the speed of pipelined computers. The situation is illustrated in Figure 1.1.

The inertia caused by the existing supercomputer installations and software applications based on pipelined computers will result in a transitional phase from the time when massively parallel computers are generally regarded to be more powerful to the time when they are dominating the supercomputer market. Today, massively parallel computers are dominating in various special purpose applications requiring high performance such as wave mechanics [GMB88] and oil reservoir simulation [SIA90].

[1]Systems with over 1000 processors were considered "massively parallel" for the purpose of Frontiers'90, The 3rd Symposium on the Frontiers of Massively Parallel Computation, October, 1990. This is consistent with earlier use of the term massive parallelism, see for instance [GMB88].

The point of intersection in Figure 1.1 is not possible to define in a precise and agreed upon manner. Danny Hillis, co-founder of Thinking Machines Corporation—the leading company in massively parallel computing, believes that the transition (in computing power) happened in 1988 [Hil90]. Other researchers would argue that the vector machines will be faster for general purpose processing also in 1995. However, the important thing is that it seems to be almost a consensus that the transition will take place in the near future.

The speed of the fastest supercomputers today is between 1 and 30 GFLOPS.[2] This is expected to increase by a factor of about 1000 by the year 2000, giving Tera (*i.e.* 10^{12}) FLOPS computers [Lun90].

Consequences for research in parallel algorithms
The continuing increase in computing power and the adoption of massively parallel computing have at least two important implications for research and education in the field of parallel algorithms:

- *We will be able to solve larger problems.* Larger problem instances increase the relevance of asymptotical analysis and complexity theory.

- *We will be able to use algorithms requiring a substantially larger number of processors.* One of the characteristics of parallel algorithms developed in theoretical computer science is that they typically require a large number of processors. These algorithms will become more relevant in the future.

To conclude, the possible proliferation of massively parallel systems will make the research done on parallel algorithms within theoretical computer science more important to the practitioners in the future. The work reported in this thesis is an attempt to learn about this research—from a practical ("engineer-like") point of view.

1.2 Summary of Contributions

The research reported in this thesis has contributed to the field of parallel processing and parallel algorithms in several ways. The main contributions

[2] 1 GFLOPS = 10^9 floating point operations per second. The precise speed depends strongly on the specific application or benchmark used in measuring the speed.

are summarised here to give a brief overview of the work and to motivate the reader for a detailed study.

- A new (or little known) direction of research in parallel algorithms is identified by raising the question: "*To what extent are parallel algorithms from theoretical computer science useful in practice?*".

- A *possibly new method of investigating this kind of parallel algorithms* is motivated and described. The method is based on evaluating the performance of *implemented* algorithms used on *finite* problems.

- The *use of the method on a large example* (Cole's parallel merge sort algorithm [Col86, Col88]) is demonstrated. This algorithm is theoretically very fast and famous within the theory community. The method and the first results achieved were presented (in a preliminary state) at *NIK'89* (Norwegian Informatics Conference) in November 1989 [Nat89]. One year after, it was presented at the *SUPERCOMPUTING'90* conference [Nat90b].

- A thorough and detailed *top down description of Cole's parallel merge sort algorithm* is given. It should be helpful for those who want a complete understanding of the algorithm. The description goes down to the implementation level, *i.e.* it includes more details than other currently available descriptions.

- Parts of Cole's algorithm which had to be made clear to be able to implement it, are described. These detailed results were presented at *The Fifth International Symposium on Computer and Information Sciences* in November 1990 [Nat90a].

- Probably the first, and possibly also the only complete parallel *implementation of Cole's parallel merge sort* algorithm have been developed. It shows that the algorithm may be implemented within the claimed complexity bounds, and gives the exact complexity constants for one possible implementation.

- A straightforward implementation of Batcher's $O(\log^2 n)$ time bitonic sorting [Bat68] on the CREW PRAM simulator is found to be faster than Cole's $O(\log n)$ time sorting algorithm as long as the number of items to be sorted n is less than 10^{21}. This shows that Cole's algorithm is without practical value.

- Prototype versions of the necessary *tools for doing this kind of algorithm evaluation* have been developed and documented. The tools have been used extensively for the work reported in this thesis, and they are currently being used for continued research in the same direction [Hag91].

- The algorithms which are studied are implemented as *high-level synchronous MIMD programs.* This programming style is consistent with early and central (theoretical) work on parallel algorithms, however it is seldom found in the current literature. Synchronous MIMD programming seems to give remarkably *easy parallel programming.* These aspects of the work were presented at the *Workshop on the Understanding of Parallel Computation* in Edinburgh, July 1990 [Nat90c].

- The prototype tools for high-level synchronous MIMD programming have been used in connection with *teaching of parallel algorithms.* They have been used for several student projects in the courses "Highly Concurrent Systems" and "Parallel Algorithms" given by the Division of Computer Systems and Telematics at the Norwegian Institute of Technology.

- As background material, the thesis provides an *introduction to parallel complexity theory*, a field which has been one of the main inspirational sources of this work. It does also contain a discussion of fundamental topics such as *Amdahl's law, speedup* and *problem scaling* which have been widely discussed during the last few years.

1.3 Motivation

This section describes how the main goal for the reported work arose from a study of (sequential) complexity theory and contemporary literature on parallel algorithms.

Two worlds of parallel algorithms
There are at least two main directions of research in parallel algorithms. In theoretical computer science parallel algorithms are typically described in a high-level mathematical notation for abstract machines, and their performance are typically analysed by assuming infinitely large problems. On the other side we have more practically oriented research, where implemented

algorithms are tested and measured on existing computers and finite problems.

The main differences between these two approaches are also reflected in the literature. One of the first general textbooks in parallel algorithms is *Designing Efficient Algorithms for Parallel Computers* by Michael Quinn [Qui87]. It gives a good overview of the field, and covers both practical and theoretical results. A more recent textbook, *Efficient Parallel Algorithms* by Gibbons and Rytter [GR88], reports a very large research activity on parallel algorithms within theoretical computer science. Comparing these two books, it seems clear that parts of the theoretical work are not clearly understood and sometimes neglected by the practitioners.

Can the theory be used in practice?

During the spring of 1988 I taught a course on parallel algorithms based on Quinn's book and other practically oriented literature. Later that year I did a detailed study of the fundamental book on complexity theory by Garey and Johnson [GJ79]. I found it difficult but also very fascinating. It led me to some papers about parallel complexity theory, and suddenly a large "new" area of parallel algorithms opened up. This area contained topics such as Nick's Class of very fast algorithms and a theory about inherently serial problems. However, I was a bit surprised by the fact that these very fundamental topics were seldomly mentioned by the practitioners.

My curiosity was further stimulated when I got a copy of the book by Gibbons and Rytter. This book showed the existence of a large and well-established research activity on theoretically very fast parallel algorithms. However, little was said about the practical value of the algorithms.

It then seemed natural to ask the following questions: Are the theoretically fast parallel algorithms simply not known by the practitioners, is the case that they are not understood, or are they in general judged as without practical value?

These questions led to a curiosity about the possible practical use of parallel algorithms and other results from theoretical computer science. I wanted to learn about the border between theoretical aspects *with* and *without* practical use, as illustrated in Figure 1.2. Contributions in this direction might lessen the gap between theory and practice within parallel processing. In January 1990, the importance of this goal was confirmed in the proceedings of the *NSF – ARC Workshop on Opportunities and Constraints of Parallel Computing* [San89a]. There, several of the more prominent re-

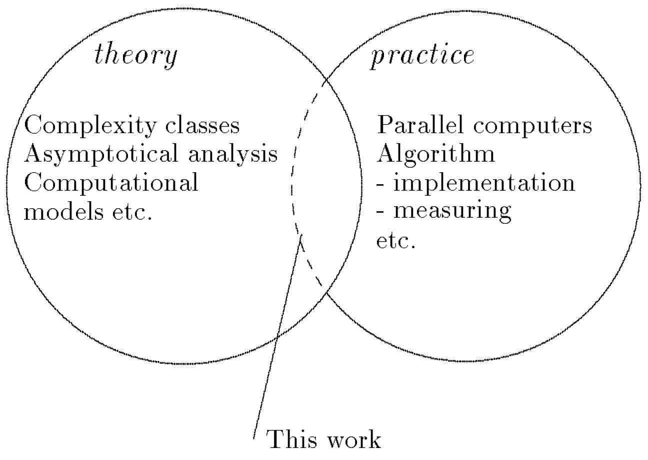


Figure 1.2: The main goal: To learn about the possible practical value of parallel algorithms from theoretical computer science.

searchers in parallel algorithms argue that bridging the gap between theory and practice in parallel processing should be one of the main goals for future research.

1.4 Background

This section describes some of the problems encountered when trying to evaluate the practical value of parallel algorithms from theoretical computer science. The discussion leads to the proposed method for doing such evaluations.

1.4.1 Parallel Complexity Theory—A Rich Source of Parallel Algorithms

In the past years there has been an increasing interest in parallel algorithms. In the field of parallel complexity theory the so called *Nick's Class* (*NC*) has been given much attention. A problem belonging to this complexity class may be solved by an algorithm with polylogarithmic[3] running time and a

[3] I.e. of $O(\log^k(n))$ where k is a constant and n is the problem size. See also Section 2.1.

polynomial number of processors. Many parallel algorithms have recently been presented to prove membership in this class for various problems. This kind of algorithms are frequently denoted *NC*-algorithms. Although the main motivation for these new algorithms often is to show that a given problem is in *NC*, the algorithms may also be taken as proposals for parallel algorithms that are "fast in practice".

In the search for fast parallel algorithms on existing or future parallel computers—the abovementioned research provides a *rich source* for ideas and hints.[4]

1.4.2 The Gap Between Theory and Practice

Unfortunately, it may be very hard to assess the practical value of *NC*-algorithms. Some of the problems are:

Asymptotical analysis
Order–notation (see Section 2.1.2) is a very useful tool when the aim is to prove membership of complexity classes or if asymptotic behaviour for some other reason is "detailed enough". It makes it possible to describe and analyse algorithms at a very high level. However, it also makes it possible to hide (willingly or unwillingly) a lot of details which cannot be omitted when algorithms are compared for "realistic" problems of finite size.

Unrealistic machine model
Few theoretical algorithms are described as implementations on real machines. Their descriptions are based on various computational models. A computational model is synonymous to an *abstract machine*. One such model is the CREW PRAM (Concurrent Read Exclusive Write Parallel Random Access Machine) model, see Chapter 3.

Details are ignored
Most *NC*-algorithms originating from parallel complexity theory are presented at a very high level and in a compact manner. One reason for this is probably that parallel complexity theory is a field that to a large extent overlaps with mathematics—where the elegance and advantages of compact

[4]Many of these algorithms may be found in proceedings from conferences such as *FOCS* and *STOC*, and in journals such as *SIAM Journal on Computing* and *Journal of the ACM.* (FOCS = IEEE Symposium on Foundations of Computer Science. STOC = ACM Symposium on Theory of Computing).

descriptions are highly appreciated.[5] A revised and more complete description of many of these algorithms may often be found, a year or two after the conference where it was presented, in journals such as *Journal of the ACM* or *SIAM Journal on Computing.* These descriptions are generally a bit more detailed, but they are still far from the "granularity" level that is required for implementation.[6] There are certainly several advantages implied by using such high level descriptions. It is probably the best way to present an algorithm to a reader who is *starting* on a study of a specific algorithm. However, there is evidently a large gap between such initial studies and implementation of an algorithm.

1.4.3 Traditional Use of the CREW PRAM Model

The certainly most used [Agg89, Col89, Kar89, RS89] theoretical model for expressing parallel algorithms is the P-RAM (Parallel RAM, also called CREW PRAM) proposed by Fortune and Wyllie in 1978 [FW78], and discussed in Section 3.1.1. Its simplicity and generality makes it possible to concentrate on the algorithm without being distracted by the obstacles caused by describing it for a more specific (and realistic) machine model. Its synchronous operation makes it easy to describe and analyse programs for the model.

This is exactly what is needed in parallel complexity theory, where the main focus is on the parallelism inherent in a problem.

1.4.4 Implementing On an Abstract Model May be Worthwhile

Implementing algorithms on an abstract machine may be regarded as a paradox. One of the reasons for using abstract machines has traditionally been a wish to avoid implementation details.

However, implementing parallel algorithms on a CREW PRAM model will provide a deeper understanding to lessen the gap between theory and practice. The following may be achieved by implementing a "theoretical" parallel algorithm on a CREW PRAM model:

[5] Another reason may be found in the call for papers for the 30'th FOCS Symposium; A strict limit of 6 pages was enforced on the submitted papers. Considering the complexity of most of these algorithms, a compact description is therefore a necessity.

[6] As an example, compare the description of Cole's parallel merge sort algorithm [Col86] and [Col88] with Section 4.2.

- *A deeper understanding.* Implementation enforces a detailed study of all aspects of an algorithm.

- *Confidence in your understanding.* Correctness verification of large parallel programs is generally very difficult. In practice, the best way of getting confident of one's own understanding of a complicated algorithm is to make an implementation that works correctly. Of course, testing is no correctness proof, but elaborate and systematic testing may give a larger degree of confidence than program verification (which also may contain errors).

- *A good help in mastering the complexity involved in implementing a complicated parallel algorithm on a real machine.* A CREW PRAM implementation may be a good starting point for implementing the same algorithm on a more realistic machine model. This is particularly important for complicated algorithms. Going directly from an abstract mathematical description to an implementation on a real machine will often be too large a step. The existence of a CREW PRAM implementation may reduce this to two smaller steps.

- *Insight into the practical value of an algorithm.* How can an unrealistic machine model be used to answer questions about the practical value of algorithms? It is important here to differentiate between negative and positive results. If a parallel algorithm A shows up to be inferior as compared with sequential and/or parallel algorithms for more realistic models, the use of an *unrealistic* parallel machine model for executing A will only *strengthen* the conclusion that A is inferior. On the other hand, a parallel algorithm can not be classified as a better alternative for practical use if it is based on a less realistic model.

These advantages are independent of whether a parallel computer that resembles the CREW PRAM model ever will be built.

A New Direction for Research on Parallel Algorithms?

To summarise, the practitioners measure and evaluate implemented algorithms for solving finite problems on existing computers, while the theorists are analysing the performance of high-level algorithm descriptions "executed" on abstract machines for "solving" infinitely large problems.

The approach presented in this thesis is to evaluate *implemented* algorithms executed on an *abstract machine* for solving *finite* problems. In Figure

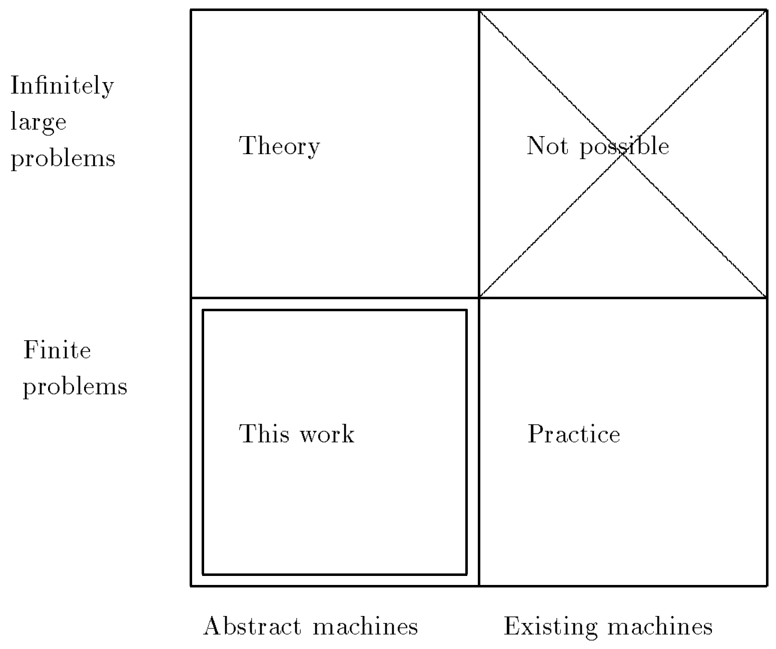


Figure 1.3: How the approach of algorithm evaluation presented in this thesis can be viewed as a compromise between the two main approaches used by the theory and practice communities.

1.3 this is identified as a third approach for evaluating parallel algorithms. It is unlikely that the work reported here is the first of this kind, so I will not claim that the approach for evaluating the practical value of theoretical parallel algorithms is new. However, I have not yet found similar work. To be better prepared for evaluating computer algorithms of the future, this direction of research should in my view be given more attention.

Chapter 2

Parallel Algorithms: Central Concepts

The first half of this chapter presents some of the more important concepts on the theory side, mainly from *parallel complexity theory*. The second half is devoted to issues that are central when *evaluating parallel algorithms* in practice.

Most of the material in this chapter is introductory in nature and provides the necessary background for reading the subsequent chapters. Parts of the text, such as Section 2.2.3 on Amdahl's law and Section 2.2.2.2 on superlinear speedup do also attempt to clearify topics about which there have been some confusion in recent years.

2.1 Parallel Complexity Theory

" The idea met with much resistance. People argued that faster computers would remove the need for asymptotic efficiency. Just the opposite is true, however, since as computers become faster, the size of the attempted problems becomes larger, thereby making asymptotic efficiency even more important."

John E. Hopcroft in his Turing Award Lecture *Computer Science: The Emergence of a Discipline* [Hop87].

This section presents those aspects of parallel complexity theory that are most relevant for researchers interested in parallel algorithms. It does not claim to be a complete coverage of this large, difficult and rapidly expanding field of theoretical computer science. Most of the concepts are presented in a relatively informal manner to make them easier to understand. The reader should consult the references for more formal definitions.

Not *all* of the material is used in the following chapters, it is mainly provided to give the necessary background, but also to introduce the reader to very central results and theory that to a large extent are unknown among the practitioners. Parallel complexity theory is a relatively new and very fascinating field—it has certainly been the main inspiration for my work.

2.1.1 Introduction

The study of the amount of computational resources that are needed to solve various problems is the study of *computational complexity.* Another kind of complexity, which has been given less attention within computer science, is descriptive, or *descriptional complexity.* An algorithm which is very complex to describe has a high descriptional complexity, but may still have a low computational complexity [Fre86a, WW86]. In this thesis, when I use the term complexity I mean *computational* complexity.

The resources most often considered in computational complexity are running time or storage space, but many other units may be relevant in various contexts. [1]

The history of computational complexity goes back to the work of Alan Turing in the 1930's. He found that there exists so called undecidable problems that are so difficult that they can not be solved by *any* algorithm. The first example was the *Halting Problem* [GJ79]. However, as described by two of the leading researchers in the field, Stephen A. Cook and Richard Karp [Coo83, Kar86], the field of computational complexity did not really start before in the 1960's, and the terms complexity and complexity class were first defined by Juris Hartmanis and Richard Stearns in the paper *On the computational complexity of algorithms* [HS65].

Complexity theory deals with computational complexity. The most central concept here is the notion of *complexity classes.* These are used to classify problems according to how difficult or hard they are to solve. The most difficult problems are the undecidable problems first found by Turing. The most well-known complexity class is the class of *NP-complete problems.* Other central concepts are *upper* and *lower bounds* on the amount of computing resources needed to solve a problem.

[1]As an example, the total number of messages (or bits) exchanged between the processors is often measured for evaluating algorithms in distributed systems [Tiw87].

Parallel complexity theory deals with the same issues as "traditional" complexity theory for sequential computations, and there is a high degree of similarity between the two [And86]. The main difference is that complexity theory for parallel computations allows algorithms to use several processors in parallel to solve a problem. Consequently, other computing resources such as the degree of parallelism and the degree of communication among the processors are also considered. The field emerged in the late 1970's, and it has not yet reached the same level of maturity as complexity theory for sequential computations.

A highly referenced book about complexity theory is COMPUTERS AND INTRACTABILITY, *A Guide to the Theory of NP–Completeness* by Garey and Johnson [GJ79]. It gives a very readable introduction to the topic, a detailed coverage of the most important issues with many fascinating examples, and provides a list of more than 300 *NP*-complete problems. A more compact "survey" is the Turing Award Lecture *An Overview of Computational Complexity* given by Stephen A. Cook [Coo83]. The Turing Award Lectures given by Hopcroft and Karp [Hop87, Kar86] are also very inspiring.

Ian Parberry's book *Parallel Complexity Theory* [Par87] gives a detailed description of most parts of the field, but Chapter 1 of Richard Anderson's Ph.D. Thesis *The Complexity of Parallel Algorithms* [And86] is perhaps more suitable as introductory reading for computer scientists.

2.1.2 Basic Concepts, Definitions and Terminology

To be able to discuss complexity theory at a minimum level of precision we must define and explain some of the most central concepts. This section may be skipped by readers familiar to algorithm analysis and complexity.

Problems, algorithms and complexity

Garey and Johnson [GJ79] define a *problem* to consist of a parameter description and a specification of the properties required of the solution. The description of the *parameters* is general. It explains their "nature" (structure), not their values. If we supply one set of values for all the parameters we have a *problem instance*. As an example, consider sorting. A formal description of the sorting problem might be

SORTING
PARAMETER DESCRIPTION: A sequence of items $S_1 = \langle a_1, a_2, \ldots, a_n \rangle$ and a total ordering $\leq$ which states whether $(a_j \leq a_k)$ is true

for any pair of items (a_j, a_k) from the set S_1.
SOLUTION SPECIFICATION: A sequence of items $S_2 = \langle a_{k_1}, a_{k_2}, \ldots, a_{k_n} \rangle$ which contains the same items as S_1 and satisfies the given total ordering, *i.e.* $a_{k_1} \leq a_{k_2} \leq \ldots \leq a_{k_n}$.

If we restrict ourself to the set of integers, which are totally ordered by our standard interpretation of $\leq$, any finite specific set of finite integers makes one instance of the sorting problem.

For most problems there exist some instances that are easy to solve, and others that are more difficult to solve. The following definition is therefore appropriate.

Definition 2.1 (Algorithm) *([GJ79] p. 4)*
An algorithm is said to **solve a problem** Π if that algorithm can be applied to any instance I of Π and is guaranteed always to produce a solution for that instance I. $\Box$

We will see that some algorithms solve all problem instances equally well, while other algorithms have a performance that is highly dependent on the characteristics of the actual instance.[2]

In general we are interested in finding the best or "most efficient" algorithm for solving a specific problem. By *most efficient* one means the minimum consumption of *computing resources*. Execution time is clearly the resource that has been given most attention. This is reasonable for sequential computations—especially today when most computers have large memories. For parallel computations other resources will often be crucial. Most important in general are the number of processors used and the amount of communication.

The time used to run an algorithm for solving a problem usually depends on the *size of the problem instance*. A tool for comparing the efficiency of various algorithms for the same problem should ideally be a yardstick which is independent of the problem size.[3] However, the relative performance of two algorithms will most often be significantly different for various sizes of the problem instance. This fact makes it natural to use a "judging tool" which has the size of the problem instance as a parameter. A *time complexity function* is such a tool and perhaps the most central concept in complexity theory.

[2]As an example, consider the concept of *presortedness* in sorting algorithms [Man85].
[3]As we shall see in Section 2.1.4, complexity classes provide this kind of "problem size independent comparison".

Definition 2.2 (Time complexity function)
A time complexity function for an algorithm is a function of the size of the problem instances that, for each possible size, expresses the largest number of basic computational steps needed by the algorithm to solve any possible problem instance of that size. □

It is common to represent the size of a problem instance as a measure of the "input length" needed to encode the instance of the problem. In this thesis, I will use the more informal term *problem size*. (See the discussion of input length and encoding scheme at pages 5–6 and 9–11 of [GJ79].) For most sorting algorithms, it is satisfactory to represent the size of the problem instance as the number of items to be sorted, denoted n. For simplicity, we will use n to represent the problem size of *any* problem for the rest of this section.

Worst case, average case and best case

Definition 2.2 leads to another central issue—the difference between worst case, average case and best case performance. Even if we restrict to problem instances of a given fixed size, there may exist easy and difficult instances. Again, consider sorting. Many algorithms are much better at sorting a sequence of n numbers which to a large degree are in sorted order than a sequence in total unorder [McG89]. The definition describes that a time complexity function represents an upper bound for all possible cases, and a more explicit term would therefore be *worst case* time complexity function. The worst case function provides a guarantee and is certainly the most frequently used efficiency measure.[4] If not otherwise stated, this is what we mean by a time complexity function in this thesis.

Best case time complexity functions might be defined in a similar way. *Average case complexity functions* are of obvious practical importance. Unfortunately, they are often very difficult to derive.

Definition 2.2 states that time is measured in "basic computational steps". It is therefore necessary to do assumptions about the underlying computational model that executes the algorithm. This important topic is treated in Section 2.1.3. For the following text, it is appropriate to think of a basic computational step as one machine instruction.

Order notation and asymptotic complexity

There is an infinite number of different complexity functions. The concept

[4] At least in the theory community.

Table 2.1: Examples on the use of big-O (order notation)

Algorithm	complexity function	order expression
A	$10n + 2n^2$	$O(n^2)$
B	$40n + 10n^2$	$O(n^2)$
C	$3n + 120n \log n$	$O(n \log n)$
D	400	$O(1)$

is therefore not an ideal tool to make *broad* distinctions between algorithms. We need an abstraction of the (exact) complexity functions which makes them more convenient to describe and discuss at a higher level and easier to compare, without loosing their most important information. *Order notation* has shown to be a successful abstraction of this kind. When we say that $f(x) = O(g(x))$ we mean, informally, that f grows at the same rate or more slowly than g when x is very large.[5]

Definition 2.3 (Big-O) *([Wil86] p. 9)*
$f(x) = O(g(x))$ if there exist positive constants c and x_0 such that $|f(x)| < cg(x)$ for all $x > x_0$. $\square$

The order notation provides a compact way of representing the *growth rate* of complexity functions. This is illustrated in Table 2.1 where we have shown the exact complexity functions and their corresponding order expressions for four artificial algorithms.[6] Note that it is common practice to include only the fastest growing term. (Algorithm D is a so called *constant time* algorithm whose execution time does not grow with the problem size.)

It is important to realise that order expressions are most precise when they represent *asymptotic complexity*, *i.e.* the problem size n becomes infinitely large. (The "error" introduced by omitting the low order terms[7] approaches zero as $n \to \infty$.) Note also that we omit the multiplicative constants—algorithms A and B are regarded as equally efficient. This is

[5] $f(x) = O(g(x))$ is read as "$f(x)$ is of order (at most) $g(x)$", or "$f(x)$ is big-oh of $g(x)$".

[6] Consider algorithm C. It would be correct to say that its complexity function is $O(n^2)$ since n^2 grows faster than $n \log n$. However, that would be a weaker statement and there is in general no reason for "saying less than we know" in this context.

[7] As done when describing A as $O(n^2)$ instead of $O(n + n^2)$.

essential for making order notation robust to changes of technology and implementation details.

Upper and lower bounds

A substantial part of computer science research consists of designing and analysing new algorithms that in some sense[8] are more efficient than those previously known. Such a result is said to establish an *upper bound.* Most upper bounds focus on the execution time, they describe asymptotic behaviour, and they are commonly expressed by using the Big-O notation.

Definition 2.4 (Upper bound) *([Akl85] p. 3)*
An upper bound (on time) $U(n)$ is established by the algorithm that, among all known algorithms for the problem, can solve it using the least number of steps in the worst case. $\square$

Several sequential algorithms exist that sort n items in $O(n \log n)$ time. One example is the Heapsort algorithm invented by J. Williams [Wil64]. Consequently, we know that sorting can be done *at least that fast* on sequential computers. Each single of these algorithms implies the existence of an upper bound of $O(n \log n)$, but note that the order notation is too rough to assess which single algorithm corresponds to the upper bound.

Lower bounds are much more difficult to derive, as expressed by Cook in [Coo83]: *"The real challenge in complexity theory, and the problem that sets the theory apart from the analysis of algorithms, is proving lower bounds on the complexity of specific problems.“*.

Definition 2.5 (Lower bound) *([Akl85] p. 3)*
A lower bound (on time) $L(n)$ tells us that no algorithm can solve the problem in fewer than $L(n)$ steps in the worst case. $\square$

While an upper bound is a result from analysing a single algorithm—a lower bound gives information about all algorithms that have been or may be designed to solve a specific problem. If a lower bound $L(n)$ has been proved for a problem Π, we say that the *inherent complexity* of Π is $L(n)$.

A lower bound is of great practical importance when designing algorithms since it clearly states that there is no reason for trying to design algorithms that would be more efficient.

[8]The possible variations in factors such as computational model, assumptions about the input (size and characteristics), and charging of the different computational resources, imply the existence of several algorithms that all in some way are most efficient for solving a (general) problem.

The influence of assumptions is illustrated by the following lower bound given by Akl in [Akl85]. This bound is trivial but highly relevant in many practical situations. See also *A "Zero-Time" VLSI Sorter* by Miranker et al. [MLK83].

> If input and/or output are done sequentially, then every parallel sorting algorithm requires $\Omega(n)$ time units.

It is common to use big-omega, Ω, to express such lower bounds. $\Omega(n)$ in this context is read as "omega of n" or "of order at least n". It can informally[9] be perceived as the opposite of 'O'. As done by several authors (Harel [Har87], Garey and Johnson [GJ79]) we will stick to big-O notation for expressing orders of magnitude.[10]

An *algorithm* with running time that matches the lower bound is said to be (time) *optimal.*

2.1.3 Models of Parallel Computation

> "Thus, even in the realm of uniprocessors, one lives happily with high-level models that often lie. This should cool down our ambition for "true to life", realistic multiprocessor models."
>
> Marc Snir in *Parallel Computation Models—Some Useful Questions* [Sni89].

While the RAM (random access machine) computational model introduced by Aho, Hopcroft and Ullmann in [AHU74] have been dominating for sequential computations we are still missing such a unifying model for parallel processing [Val90, San89b]. A very large number of models of parallel computation have been proposed. The state is further cluttered up by the fact that various authors in the field often use different names on the same model. They also often have different opinions on how a specific model should operate [San89b].

This section does not aim at giving a complete survey of all models. It starts by shortly mentioning some classification schemes and surveys, and then introduces the P-RAM model and its numerous variants.

[9] In fact the 'Ω' is the "negation" of 'o' which is a more precise variant of 'O'. ('Θ', often read as "of order exactly" is still more precise.) Exact definitions of the five symbols 'O', 'Ω', 'Θ', 'o' and '$\sim$' that are used to compare growth rates may be found in [Wil86].

[10] This is to avoid introducing excessive notation. $\Omega(f(n))$ may be expressed as of order at least $O(f(n))$.

The P-RAM model is the most central concept in this thesis, and is therefore thoroughly described in Chapter 3. The section ends with a discussion of important general properties of models, and some comments on a very interesting model recently proposed by Leslie Valiant [Val90].

2.1.3.1 Computing Models—Taxonomies and Surveys

There has been proposed a very large number of models for parallel computing ranging from realistic, message passing, asynchronous multiprocessors—to idealised, synchronous, shared memory models. These models and all the various kinds of parallel computers that have been built form a diverse, and complicated dynamic picture. This "mess" has motivated the development of numerous taxonomies and classification schemes for models of parallel computing and real parallel computers.

Flynn's taxonomy
Without doubt, the most frequently used classification of (parallel) computers is Flynn's taxonomy presented by Michael Flynn in 1966 [Fly66]. The taxonomy classifies computers into four broad categories:

SISD *Single Instruction stream – Single Data stream*
The von Neumann model, the RAM model, and most serial (*i.e.* uniprocessor) computers fall into this category. A single processor executes a single instruction stream on a single stream of data. However, SISD computers may use parallelism in the form of pipelining to speed up the execution in the single processor. The CRAY-1, one of the most successful supercomputers is classified as a SISD machine by Hwang and Brigg [HB84].

SIMD *Single Instruction stream – Multiple Data stream*
Models and computers following this principle have several processing units each operating on its own stream of data. However, all the processing units are executing the same instruction from one instruction stream, and they may therefore be realised with a single control unit. Array processors are in this category. Classical examples are ILLIAC IV, ICL DAP and MPP [HB84]. The Connection Machine [Hil87, HS86] is also SIMD. Most SIMD computers operates synchronously using a single global clock.

Many authors classify the PRAM model as SIMD, but this is not consistent with the original papers defining that model (see Section

3.2.1).

MISD *Multiple Instruction stream – Single Data stream*
This class is commonly described to be without examples. It can be perceived as several computers operating on a single data stream as a "macro-pipeline" [HB84]. Pipelined vector processors, which commonly are classified as SISD, might also be put into the MISD class [AG89].

MIMD *Multiple Instruction stream – Multiple Data stream*
These machines have several processors each being controlled by its own stream of instructions and each operating on its own stream of data. Most multiprocessors fall into this category. The processors in MIMD machines may communicate through a shared global memory (tightly coupled), or by message passing (loosely coupled). Examples are Alliant, C.mmp, CRAY-2, CRAY X-MP, iPSC, Ncube [AG89, HJ88].

Some authors use the term MIMD almost as synonymous with asynchronous operation [Akl89, Ste90, Dun90]. It is true that most MIMD computers which have been built operate asynchronously, but there is nothing wrong with a synchronous MIMD computer (See also Section 3.2.1).

In practice Flynn's taxonomy gives only two categories of parallel computers, SIMD and MIMD. Consequently, it has long been described as too coarse [Sny88, AG89].

Duncan's taxonomy and other classification schemes

Ralph Duncan [Dun90] has very recently given a new taxonomy of parallel computer architectures. It is illustrated in Figure 2.1. Duncan's taxonomy is a more detailed classification scheme than Flynn's taxonomy. It is a high-level practically (or machine) oriented scheme—well suited for classifying most contemporary parallel computers into a small set of classes. The reader is referred to Duncan's paper for more details.

Many other taxonomies have been proposed. Basu [Bas87] has described a taxonomy which is tree structured in the same way as Duncan's taxonomy, but more detailed and systematic. Other classification schemes have been given by Händler (see for instance [HB84], page 37), Feng (see [HB84], page 35), Kuck (see [AG89], page 112), Snyder [Sny88], and Treleaven (see [AG89], page 113).

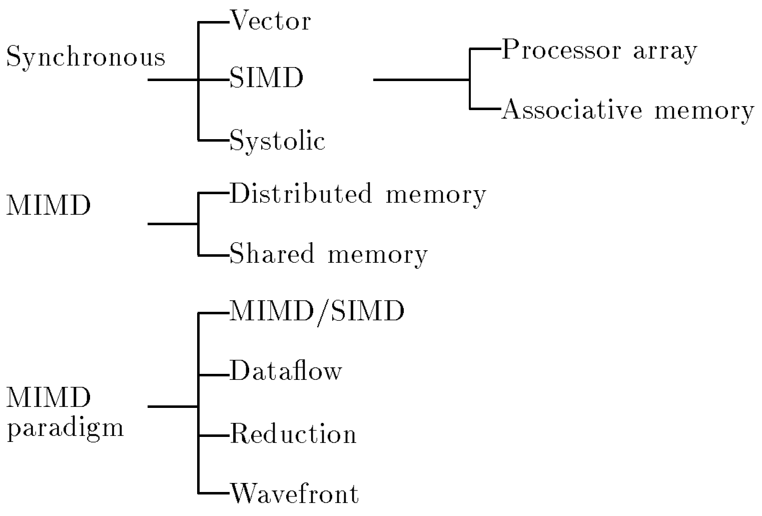


Figure 2.1: Duncan's taxonomy of parallel computer architectures [Dun90].

Surveys
A good place to start reading about existing computers and models for parallel processing is Chapter 2 of *Designing Efficient Algorithms for Parallel Computers* written by Quinn [Qui87].[11] Surveys are also given by Kuck [Kuc77], Mikloško and Kotov [MK84], Akl [Akl89], DeCegama [DeC89], and Almasi and Gottlieb [AG89]. H. T. Kung, who is well known for his work on systolic arrays, has written a paper *Computational models for parallel computers* [Kun89] which advocates the importance of computational models. The scope of the paper is restricted to 1D (*i.e.* linear) processor arrays, but Kung describes 9 different computational models for this rather restricted kind of parallel processing. This illustrates the richness of parallel processing.

Theoretical models
The taxonomies and surveys mentioned so far are mainly representing parallel computers that have been built. Some of the more realistic models for parallel computation may be classified with these taxonomies. However, many of the more theoretic models do not fit into the schemes. Again, we

[11]For more detailed descriptions of the most important parallel computers that have been made see Hwang and Briggs [HB84], Hockney and Jesshope [HJ88].

notice the gap between theory and practice.

The paper *Towards a complexity theory of synchronous parallel computation* written by Stephen Cook [Coo81] describes many of the most central models for parallel computation that are used in the theory community. In Cook's terminology these are uniform circuit families, alternating Turing machines, conglomerates, vector machines, parallel random access machines, aggregates and hardware modification schemes. The paper is highly mathematical, and the reader is referred to the paper for more details.[12] *A Taxonomy of Problems with Fast Parallel Algorithms*, also written by Cook [Coo85], is mainly focusing at algorithms but does also give an updated view of the most important models. The paper gives an extensive list of references to related work.

The first part of *Routing, Merging and Sorting on Parallel Models of Computation* by Borodin and Hopcroft [BH85], and *Parallel Machines and their Communication Theoretical Limits* by Reischuk [Rei86] give overviews with a bias which are closer to practical computers. More specifically, they are giving more emphasis on the shared memory models which are easier to program but less mathematical convenient.

Perhaps the most "friendly" *introduction* to theoretical models for parallel computation for computer scientists is Chapter 2 of James Wyllie's PhD Thesis *The Complexity of Parallel Computations* [Wyl79]. Wyllie motivates and describes the P-RAM model which probably is the most used theoretical model for expressing parallel algorithms. It is described below.

2.1.3.2 The P-RAM Model and its Variants

> "Most parallel algorithms in the literature are designed to run on a PRAM."
>
> Alt, Hagerup, Mehlhorn and Preparata in *Deterministic Simulation of Idealised Parallel Computers on More Realistic Ones*, [AHMP86].

> "The standard model of synchronous parallel computation is the P-RAM."
>
> Richard Anderson in *The Complexity of Parallel Algorithms*, [And86].

[12] I have not studied all the details of this and some of the other mathematical papers which are referenced. The purpose of the short description and the references is to introduce the material and give exact pointers for further reading.

"The parallel random-access machine (PRAM) is by far the most popular model of parallel computation."

Bruce Maggs in *Beyond Parallel Random-Access Machines*, [Mag89].

The most popular model for expressing parallel algorithms within theoretical computers science is the P-RAM model [Agg89, Col89, Kar89, RS89].

The Original P-RAM of Fortune and Wyllie

The *parallel random access machine* (P-RAM) was first presented by Steven Fortune and James Wyllie in [FW78]. It was further elaborated in Wyllie's well-written Ph.D. thesis *The Complexity of Parallel Computations* [Wyl79]. The P-RAM is based on random access machines (RAMs) operating in parallel and sharing a common memory. Thus it is in a sense the model in the world of parallel computations that corresponds to the RAM (Random Access Machine) model that certainly is the prevailing model for sequential computations. The processors operate synchronously. The connection to the RAM model should not be a surprise since John Hopcroft was the thesis advisor of James Wyllie.

Today, a large number of variants of the P-RAM model are being used. The original P-RAM model of Fortune and Wyllie, which is the main underlying concept of this work, is described in Chapter 3. Below we describe its main variants, and mention some of the other names that have been used on these.

The EREW, CREW and CRCW Models

Today, the mostly used name on the original P-RAM model is probably CREW PRAM [SV84]. CREW is an abbreviation for the very central concurrent read exclusive write property.[13] The main reason for this name is the possibility to explicitly distinguish the model from the two other variants—EREW PRAM and CRCW PRAM.

The EREW PRAM does not allow concurrent read from the global memory. It may be regarded as more realistic than the CREW PRAM, and some authors present algorithms in two variants—one for the CREW PRAM and another for the EREW PRAM model. The EREW PRAM algorithms are in general more complex, see for instance [Col88].

[13]This means that several processors may at the same time step read the same variable (location) in global memory, but they may not write to the same global variable simultaneously.

The CRCW PRAM allows concurrent writing in the global memory and is less realistic than the CREW PRAM. It exists in several variants, mainly differing in how they treat simultaneous "writes" to the same memory location (see below). Reischuk [Rei86] discusses the power of concurrent read and concurrent write. The ERCW variant, which is the fourth possibility, is in general not considered because it seems more difficult to realise simultaneous writes than simultaneous reads [Rei86].

The CREW PRAM model has probably become the most popular of these variants because it is the most powerful model which also is "well-defined"—the CRCW variant exists in many subvariants.

The various CRCW models

One of the problems with the CRCW PRAM model is the use of several definitions for the semantics of concurrent writing to the global memory. The main subvariants for the handling of two or more simultaneous write operations[14] to the same global memory cell are:

1. *Common value only.* An arbitrary number of processors may write to the same global memory location provided that they all write the same value. Writing different values is illegal. [Fei88, Akl89, BH85, Rei86]

2. *Arbitrary winner.* One arbitrary of the processors which are trying to write to the same location succeeds. The value attempted written by the other processors are lost. (This gives nondeterministic operation). [Fei88, Rei86, BH85]

3. *Ordered.* The processors are ordered by giving them different priorities, and the processor with highest priority wins. [Fei88, Rei86, Akl89, BH85]

4. *Sum.* The sum of the quantities written is stored. [Akl89]

5. *Maximum.* The maximum of the values written is stored. [Rei86]

6. *Garbage.* The resulting value is undefined. [Fei88]

PRAC, PRAM, WRAM etc.—Terminology

The various variants of the PRAM model have been used under several

[14]This is often called a write conflict.

different names. The following list is an attempt to reduce possible confusion and to mention some other variants.

PRAC, corresponds to EREW PRAM [BH85].

PRAM, is the original P-RAM model of Fortune and Wyllie, and has often been given rather different names. In [QD84] the term MIMD-TC-R is used, and in [Qui87] SIMD-SM-R is used.

WRAM, corresponds to CRCW PRAM [BH85, GR88].

CRAM, ARAM, ORAM, and *MRAM*, have been used by Reischuk [Rei86] to denote the, respectively, *C*ommon value only, *A*rbitrary winner, *O*rdered, and *M*aximum variants of the CRCW PRAM model (see above).

DRAM, is short for *distributed random access machine* and has been proposed by Leiserson and Maggs as a more realistic alternative to the PRAM model [Mag89].

2.1.3.3 General Properties of Parallel Computational Models

Motivation

A *computing model* [15] is an idealised, often mathematical description of an existing or hypothetical computer. There are many advantages of using computing models.

- *Abstractions simplify.* When developing software for any piece of machinery a computing model should make it possible to concentrate on the most important aspects of the hardware, and hiding low-level details.

- *Common platform.* A computing model should be an agreed upon standard among programmers in a project team. This makes it easier to describe and discuss measured or experienced performance of various parts of a software system. A good model will increase software portability, and also provide a common language for research and teaching [Sni89]. (See also the paragraph below about Valiant's bridging BSP model.)

- *Analysis and performance models.* In theoretical computer science computational models have played a crucial role by providing a common base for algorithm analysis and comparison. The RAM model has

[15]Note that the literature use a wide variety of terms for this concept. Examples are computing model, computer model, computation model, computational model, and model of computation.

been used as a common base for (asymptotic) analysis and comparison of sequential algorithms. For parallel computations the P-RAM model of Fortune and Wyllie [FW78, Wyl79] has made it possible, for a typical analysis of parallel algorithms, to concentrate on a small set of well defined quantities; number of parallel steps (time), number of processors (parallelism) and sometimes also global memory consumption (space). More realistic models typically use a larger number of parameters to describe the machine, and on the other side of the specter we have special purpose performance models which are highly machine dependent and often also specific to a particular algorithm. (See for instance [AC84, MB90].)

What is the "right" model?

The selection of the appropriate model for parallel computation is certainly one of the most widely discussed issues among researchers in parallel processing. This is clearly reflected in the proceedings of the *NSF – ARC Workshop on Opportunities and Constraints of Parallel Computing* [San89a]. There are many difficult tradeoffs in this context. Some examples are easy (high-level) programming vs. efficient execution (*e.g. shared memory vs. message passing*) [Bil89], general purpose model vs. special purpose model (*e.g. P-RAM vs. machine or algorithm specific models*), easy to use vs. mathematical convenience (*e.g. P-RAM vs. "boolean circuit families"*[Coo85]), and easy to analyse vs. easy to build (*e.g. synchronous vs. asynchronous*).

Instead of arguing for some specific (type of) model to be the best, we will describe important properties of a good model. Some of these properties are overlapping, and unfortunately, several of the desired properties are conflicting.

1. *Easy to understand.* Models that are difficult to understand, or complex in some sense (for instance by containing a lot of parameters or allowing several variants) will be less suitable as a common platform. Different ways of understanding the model will lead to different use and reduced possibilities of sharing knowledge and experience. The importance of this issue is examplified by the PRAM model. In spite of being one of the simplest models for parallel computations it has been understood as a SIMD model by many researchers and as a MIMD model by others (see Section 3.2.1).

2. *Easy to use.* Designing parallel algorithms is in general a difficult

task. A good model should help the programmer to forget unnecessary low-level details and peculiarities of the underlying machine. The model should help the programmer to concentrate on the problem and the possible solution strategies—it should not *add* to the difficulties of the program design process. Simple, high level models are in general most easy to use. They are well suited for teaching and documentation of parallel algorithms. However, when designing software for contemporary parallel computers one is often forced to use more complicated and detailed models to avoid "loosing contact with the realities". Synchronous models are in general easier to program than asynchronous models. This is reflected by the fact that some authors use the term *chaotic* models to denote asynchronous models [Gib89]. Shared memory models seem to be generally more convenient for constructing algorithms [BH85, Par87].

3. *Well defined.* A good model should be described in a complete and unambigious way. This is essential for acting as a common platform. The so-called CRCW PRAM model exists in many variants (see page 26) and is therefore less popular than the CREW PRAM model.

4. *General.* A model is general if it reflects many existing machines and more detailed models. The use of general models yields more portable results. The RAM model has been a very successful general model for uni-processor machines.

5. *Performance representative.* Marc Snir has written an interesting paper [Sni89] discussing at a general level to what extent high level models lead to the creation of programs that are efficient on the real machine. Informally, good models should give a performance rating of (theoretical) algorithms (*i.e.* run on the model) that is closely related to the rating obtained by running the algorithms on a real computer [Sni89] (See also [Sny88]). In other words, the practically good algorithms should be obtained by refining the theoretically good ones.

6. *Realistic.* Many researchers stress the importance of a computing model to be feasible [Agg89, Col89, Sny88]. Models that can be realised with current technology without violating too many of the model assumptions have many advantages. Above all, they give a model performance which is representative of real performance as discussed above. With respect to feasibility, there is a great difference

between models that are based on a fixed connection network of processors and models based on the existence of a global or shared memory. The fixed connection models are much easier to realise, and in fact the best way of implementing shared memory models with current technology[BH85, RBJ88]. Unfortunately, the fixed connection models are in general regarded as more difficult to program. Similarly, asynchronous operation of the processors is realistic but generally accepted as leading to more difficult programming [Ste90]. On the other hand there are researchers arguing for more idealised and powerful models than the PRAM (which is the standard shared memory model) [RBJ88]. Note also that there are reasonable arguments for a model to be "far from" current technology [Vis89, Val90]. This is explained in the following property and in the next paragraph.

7. *Durable.* Uzi Vishkin [Vis89] argues that computing models should be robust in time in contrast to technological feasibility which rapidly keeps advancing. Again, the RAM model is an example. Changing models too often will greatly reduce the possibilities of sharing information and building on other work. However, machines will and should change—new technological opportunities continue to appear. In practice, this is an argument against realistic models such as asynchronous fixed connection models (sometimes also called message passing models). A similar view has been expressed by Leslie Valiant [Val90].

8. *Mathematical convenience.* Stephen Cook describes shared memory models as unappealing for an enduring mathematical theory due to the arbitrariness in its detailed definition. He advocates uniform Boolean circuit families as more attractive for such a theory [Coo85]. In my view, models based on circuit families are not suited for expressing large, practical parallel systems. Most algorithm designers use the PRAM shared memory model, and it should be noted that it contains two drastic assumptions which are introduced for mathematical convenience [Wyl79]. Assuming *synchronous operation* of all the processors makes the notion of "running time" well defined and is crucial for analysing time complexity of algorithms. Assuming *unbounded parallelism* (*i.e.* an unbounded number of processors is available) makes it possible to handle asymptotical complexity.

2.1.3.4 Valiant's Bridging BSP Model

Leslie Valiant has recently written a very interesting paper [Val90] were he advocates the need for a bridging model for parallel computation, and proposes the *bulk-synchronous parallel (BSP)* model as a candidate for this role.

He attributes the success of the von Neumann model of sequential computation to the fact that it has been a bridge between software and hardware. On the one side the software designers have been producing a diverse world of increasingly useful and resource demanding software assuming this model. On the other side the hardware designers have been able to exploit new technology in realising more and more powerful computers providing this model.

Valiant claims that a similar standardised *bridging model* for parallel computation is required before general purpose parallel computation can succeed. It must imply convenient programming so that the software people can accept the model over a long time. Simultaneously, it must be sufficiently powerful for the hardware people to continuously provide better implementations of the model. A realistic model will not act as a bridging model because technological improvements are likely to make it old-fashioned too early. For the same reason a bridging model should also be simple and not defined at a too detailed level. There should be open design choices allowing better implementations without violating the model assumptions [Val90].[16]

Valiants BSP model of parallel computation is defined as the combination of three attributes: a number of processing components, a message router and a synchronisation facility. The *processing components* perform processing and/or memory functions. The *message router* delivers messages between processing components. The *synchronisation facility* is able to synchronise all or a subset of the processing components at regular intervals. The length of this interval is called the *periodicity*, and it may be controlled by the program. In the interval between two synchronisations (called a superstep) processing components perform computations asynchronously. In this sense, the BSP model may be seen as a compromise between asynchronous and synchronous operation. Phillip B. Gibbons has expressed sim-

[16]An example from another field of computer science is the success of the relational model in the database world. This may be explained by the fact that the relational model has acted as a bridge between users of database systems and implementors. When proposed, the model was simple and general, high level and "advanced". Database system designers have worked hard for a long time to be able to provide efficient implementations of the relational model. (This example was pointed out to me by P. Thanisch [Tha90].)

ilar thoughts and use the term *semi-synchronous* for this kind of operation [Gib89].

A computer based on the BSP model may be programmed in many styles, but Valiant claims that a PRAM language would be ideal.[17] The programs should have so-called *parallel slackness*, which means that they are written for v virtual processors run on p physical processors, $v > p$. This is necessary for the compiler to be able to "massage" and assemble *bulk*s of instructions executed as supersteps giving an overall efficient execution.[18] Valiant's BSP model is a totally new computing concept requiring drastically new designs of compilers and runtime systems.

The BSP model can be realised in many ways and Valiant outlines implementations using packet switching networks or optical crossbars as two alternatives [Val90].

2.1.4 Important Complexity Classes

There are three levels of problems. The simplest problem is what we meet in everyday life—solving a specific instance of a problem. The next level up is met by algorithm designers—making a general receipt for solving a (subset of) all possible instances of a problem. Above that we have a metatheoretic level where the whole structure of a class of problems are studied [Fre86a]. At this highly mathematical level one is interested in various kinds of relationships between a large number of complexity classes (see for instance Chapter 7 in Garey and Johnson [GJ79]).

A *complexity class* can be seen as a practical interface between these two topmost levels. The complexity theorist classifies various problems and also extends the general knowledge of the different complexity classes, while the algorithm designer may use this knowledge once parts of his practical problem have been classified.

2.1.4.1 P, NP and NP-completeness

The most important complexity classes for sequential computations are *P*, *NP* and the class of *NP*-complete problems (often abbreviated *NPC* [GJ79],[Har87]). It is common to define complexity classes in terms of Turing

[17]Reading this at the end of the work reported in this thesis was highly encouraging—most of the practical work has been the design and use of a prototype high level PRAM language, see Chapter 3.

[18]It seems to me that Cole's parallel merge sort algorithm, described in Chapter 4, may be a good example on a program with parallel slackness.

Machines and language recognition ([GJ79, RS86]), but more informal definitions will suffice. In the discussion of "reasonable" parallelism at page 36 we will see that these complexity classes also are highly relevant for parallel computations.

Definition 2.6 (The class P)
The class P is the class of problems that can be solved on a sequential computer by a deterministic polynomial time algorithm. □

Definition 2.7 (Polynomial time algorithm) *([GJ79] p. 6)*
A polynomial time algorithm is an algorithm whose time complexity function is $O(f(n))$ for some polynomial function f, where n is the problem size. □

Definition 2.8 (Exponential time algorithm) *[GJ79] p. 6)*
An exponential time algorithm is an algorithm whose time complexity function can not be bounded by a polynomial function. □

The word "deterministic" might have been omitted from Definition 2.6 since it corresponds to our standard interpretation of what an algorithm is. The motivation for including it becomes clear when we have defined the larger class *NP* which also includes problems that can not be solved by polynomial time algorithms.

Definition 2.9 (The class NP)
"Standard formulation": The class *NP* is the class of problems that can be *solved* by a *nondeterministic* polynomial time algorithm.
"Practical formulation": The class *NP* is the class of problems which has solutions that can be *verified* (to be a solution or not) by a deterministic polynomial time algorithm. □

Note that we avoided the term reasonable computer in the standard formulation above. This is because the concept *nondeterministic algorithm* intentionally contains a so called guessing stage [GJ79] which (informally) is assumed to guess the correct solution. This feature is magical [Har87] and makes the properties of the underlying machine model irrelevant.

The most celebrated complexity class is the class of *NP-complete problems*. Informally the class can be said to contain the hardest problems in *NP*. Though it is not proved, it is generally believed that none of these problems can be solved by polynomial time algorithms. Today, close to 1000 problems are known to be *NP*-complete [Har87]. If you find an algorithm that

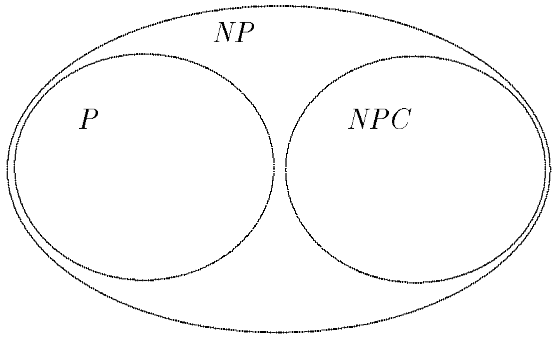


Figure 2.2: The complexity classes *NP*, *P*, and *NPC*.

solves one of these problems in polynomial time—you have really done a giant breakthrough in computer science. This should come clear from the definition of the class *NPC*.

Definition 2.10 (NP-complete problems, (NPC))
NPC is the class of *NP*-complete problems. A problem Π is *NP*-complete if *i)*: $\Pi \in NP$, and *ii)*: For any other problem Π' in *NP* there exists a polynomial transformation from Π' to Π. □

A *polynomial transformation* from A to B is a (deterministic) polynomial time algorithm for transforming (or reducing) any instance I_A of A to a corresponding instance I_B of B so that the solution of I_B gives the required answer for I_A. As a consequence (see Lemma 2.1 at page 34 of [GJ79]) it can be proved that a polynomial time algorithm for solving problem B combined with this polynomial transformation yields a polynomial time algorithm for solving A. Thus Definition 2.10 states that a polynomial time algorithm for a single *NP*-complete problem implies that all problems in *NP* can be solved in polynomial time. The relationship between the classes, *NP*, *P* and the class *NPC* containing the *NP*-complete problems, is shown in Figure 2.2.

Reading part *ii)* of Definition 2.10 one might think that much work is required to prove that a problem is *NP*-complete. Fortunately, it is not necessary to derive polynomial transformations from all other problems in *NP*. This and other aspects of *NP*-completeness will become clear in Section 2.1.4.3 at page 37 where we describe *P*-completeness and its strong similarities with *NP*-completeness.

2.1.4.2 Fast Parallel Algorithms and Nick's Class

With respect to parallelisation, the problems inside P can be divided into two main groups—the class NC and the class of P-complete problems. The class NC contains problems which can be solved by "fast" parallel algorithms that use a "reasonable" number of processors.

Definition 2.11 (Nick's class, NC)
Nick's Class NC is the class of problems that can be solved in polylogarithmic time (*i.e.* time complexity $O(\log^k n)$ where k is a constant) with a polynomial number of processors (*i.e.* bounded by $O(f(n))$ for some polynomial function f, where n is the problem size). □

NC is an abbreviation for *Nick's Class*, and is now the commonly used [Coo85] name for this complexity class, which was first identified and characterised by Nicholas Pippenger in 1979 [Pip79].

It is relatively easy to prove that a problem is in NC. Once one algorithm with $O(\log^{k_1} n)$ time consumption using no more than $O(n^{k_2})$ processors that solves the problem has been found, the membership in NC is proved. Such an algorithm is often called a *NC-algorithm*. Thus Batcher's $O(\log^2 n)$ time sorting network using $n/2$ processors (see Section 4.1.3) proves that sorting is in NC. Many other important problems are known to be in NC. Some examples are matrix multiplication, matrix inversion, finding the minimum spanning forest of a graph [Coo83], and the shortest path problem ([CM88] p. 107). The book *Efficient Parallel Algorithms* written by Gibbons and Rytter [GR88] provides detailed descriptions of a large number of NC-algorithms.

NC is robust
The *robustness* of Nick's Class is probably the main reason for its popularity. It is robust because it to a large extent is insensitive to differences between many models of parallel computation. NC will contain the same problems whether we assume the EREW, CREW or CRCW PRAM model, or some other models such as uniform circuits [Coo85, And86, Har87, GR88].[19] The robustness of NC allows us to ignore the polylogarithmic factors that separate the various models.

More refined complexity classes such as NC^1 and NC^2 may be defined within NC. NC^1 is used to denote the class of problems in NC that can be

[19]This is shown by various theoretical results describing how theoretical models may be simulated by more realistic models, see for instance [AHMP86, MV84, SV84, Upf84].

solved in $O(\log n)$ time, whereas NC^2 is the problems solvable in $O(\log^2 n)$ time. These classes are not robust to changes in the underlying model and should therefore only be used together with a statement of the assumed model. Many other subclasses within *NC* are described in Stephen Cook's article "*A Taxonomy of Problems with Fast Parallel Algorithms*" [Coo85].

"Reasonable" parallelism

Classifying a polynomial processor requirement as "reasonable" in general may need some explanation. Practitioners would for most problems say that a processor requirement that grows faster than the problem size might not be used in practice. However, in the context of complexity theory it is common practice to say that problems in *P* can be solved in "reasonable" time as opposed to problems in *NPC* that (currently) only can be solved in unreasonable (*i.e.* exponential) time. Similarly, we distinguish between reasonable (polynomially) and unreasonable (exponentially) processor requirements.

NP-complete problems can (at least in the theory)[20] be solved in polynomial time if we allow an exponential number of processors "to remove the magical nondeterminism" (see [Har87] p. 271). However, if we restrict to a "reasonable" number of processors, the *NP*-complete problems remain outside *P*.[21] In this sense the traditional complexity class *P* is very robust. *P* contains the same problems whether we assume one of the standard sequential computational models or parallel processing on a reasonable number of processors. A polynomial number of processors may be simulated with a polynomial slowdown on a single processor. This implies that all problems in *NC* must be included in the class *P*.

NC may be misleading

In theoretical computer science there has in the past years been published a large number of papers proving various (sub)problems to be in *NC*. These results are of course valuable contributions to parallel complexity theory. However, if you are searching for good *and practical* parallel algorithms, the titles of these papers may often be misleading if you do not know the terminology or do not understand the limitations of complexity theory. For

[20]It is far from clear that it can be done in practice due to inherent limitations of three-dimensional space [Coo85, Har87, Fis88].

[21]To see this, assume the opposite—a parallel algorithm using $p_1(n)$ processors that solves an *NP*-complete problem in $p_2(n)$ time, where $p_1(n)$ and $p_2(n)$ are both polynomials. Simulating this algorithm on a single processor would then give a polynomial time algorithm $(O(p_1(n) \times p_2(n)))$, the problem must be in *P* and we must have $NPC = P$.

instance "Fast Parallel Algorithm for Solving ..." may denote an algorithm with $O(\log^k n)$ time consumption with very large complexity constants which makes it slower in practice than well known and simpler algorithms for the same problem. Similarly, "Efficient Parallel Algorithm for Solving ..." may entitle a polylogarithmic time algorithm with $O(n^5)$ processor requirement resulting in a very high cost and a very *low efficiency*—if we use the definition of efficiency which is common in other parts of the parallel processing community (see page 47). It is therefore possibly more appropriate to describe problems in *NC* as "highly parallelisable" [GR88]. Further, the term *NC*-algorithm is more precise and may often be less misleading than to say that an algorithm is fast or efficient.

It has been argued that too much focus has been put on developing *NC*-algorithms [Kar89]. The search for algorithms with polylogarithmic time complexity has produced a lot of algorithms with a large (but polynomially bounded) number of processors. These algorithms typically have a very high cost compared with sequential algorithms for the same problems—and consequently low efficiency. Richard Karp has therefore proposed to define an algorithm to be efficient if its cost is within a polylogarithmic factor of the cost of the best sequential algorithm for the same problem [Kar89]. See also the recent work by Snir and Kruskal [Kru89].

2.1.4.3 Inherently Serial Problems and P-Completeness

P-completeness

Consider the problems in *P*. The subset of these which are in *NC* may be regarded as easy to parallelise, and those outside *NC* as difficult or hard to parallelise. Proving that a problem is outside *NC* (*i.e.* proving a lower bound) is however very difficult.

An easier approach is to show problems to be as least as difficult to parallelise as other problems. The notion of *P-completeness* helps us in doing this. Just as the *NP*-complete problems are those inside *NP* which are hardest to solve in polynomial time, the class of *P*-complete problems are those inside *P* which are hardest to solve in polylogarithmic time using only a polynomial number of processors.[22]

Definition 2.12 (P-complete problems, PC) *([GR88] p. 235)*
PC is the class of *P*-complete problems. A problem Π is *P*-complete if *i)*:

[22]Note that some early papers on *NP*-completeness used the term *P-complete* for those problems that are common today to denote as *NP*-complete, see for instance [SG76].

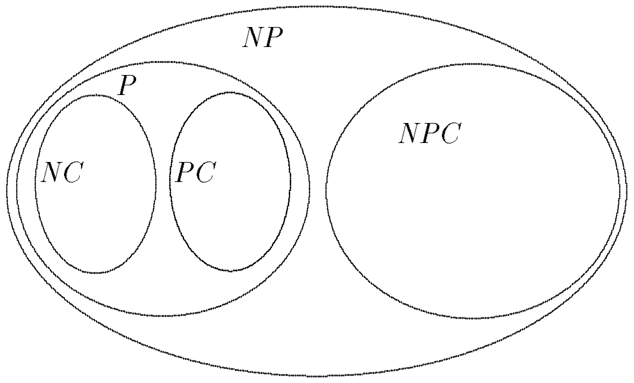


Figure 2.3: The complexity classes *NP*, *P*, *NPC*, *NC* and *PC*.

$\Pi \in P$, and *ii)*: For any other problem Π' in P there exists a *NC*-reduction from Π' to Π. □

Definition 2.13 (NC-reduction) *([GR88] p. 235)*
A *NC*-reduction from A to B (written as $A \propto_{NC} B$) is a (deterministic) polylogarithmic time bounded algorithm with polynomially bounded processor requirement for transforming (or reducing) any instance I_A of A to a corresponding instance I_B of B so that the solution of I_B gives the required answer for I_A. □

Consequently, if one single *P*-complete problem can be solved in polylogarithmic time on a polynomial number of processors—then all problems in P can be solved within the same complexity bounds using the *NC*-reduction required by the definition.

Just as there is no known proof that $P \neq NP$, nobody has been able to prove that $NC \neq P$. However, most researchers believe that the hardest parallelisable problems in P (the *P*-complete problems) are outside *NC*. A *P*-completeness result has therefore the same practical effect as a lower bound—discouraging efforts for finding *NC*-algorithms for solving the problem. The relationship between the classes, *P*, *NC* and the class *PC* containing the *P*-complete problems, is shown in Figure 2.3. The figure also shows the three main complexity classes for sequential computations.

NC-reductions

A *NC*-reduction is technically different from the transformation used in classical definitions of *P*-completeness [GR88]. As done by Gibbons and Rytter

[GR88] we will use the notion of *NC*-reductions to avoid introducing additional details about space complexity. Classical definitions use so called logarithmic space reductions, and *P*-complete problems are often termed as "log-space complete for *P*".[23] However, by using the so called *parallel computation thesis*[24] it can be shown that a log-space reduction corresponds to a *NC*-reduction, and they will be used as synonyms in the rest of the text.

A *NC*-reduction may informally be perceived as a "fast" problem reduction executable on a "reasonable" parallel model. It is interesting to note that Gibbons and Rytter [GR88] have observed that *NC*-reductions typically work very locally. For example an instance G of a graph problem may often be transformed (reduced) to a corresponding instance G' of another graph problem by transforming the local neighbourhood of every node in G. This is intuitively not surprising. There exists many examples where the ability to do computations based on local information (instead of global, central resources) significantly improves the possibilities of achieving highly parallel implementations.

A very important property of the "*NC*-reducability" relation, denoted $\propto_{NC}$, is that it is transitive [Par87, And86]. As described below, this property is central in reducing the efforts needed for proving a problem to be *P*-complete.

P-completeness proofs

Part *ii)* of Definition 2.12 of *P*-complete problems may make us believe that the work needed to prove a new problem to be *P*-complete is substantial since we must provide a *NC*-reduction from every problem known to be in *P*. Fortunately, this is not true due to the following observation.

Observation 2.1 *(Corollary 5.2.3 in [Par87])*
If A is *P*-complete, $B \in P$, and $A \propto_{NC} B$ then B is *P*-complete.

Proof: We want to show that B is *P*-complete. Since $B \in P$, what remains to show is that for every problem $\Pi \in P$ there exists a *NC*-reduction from Π to B, *i.e.* $\Pi \propto_{NC} B$. Since A is known to be *P*-complete Definition 2.12 implies $\Pi \propto_{NC} A$. The assumption $A \propto_{NC} B$ and the transitivity of $\propto_{NC}$ then imply that $\Pi \propto_{NC} B$. (From the proof of Lemma 2.3 in Garey and

[23] A *log-space reduction* is a transformation in the same sense as the polynomial transformation described at page 34 but with a space consumption which is bounded by $O(\log n)$ [GJ79, And86].

[24] This result informally states that sequential memory space is equivalent to parallel time up to polynomial differences [Har87, And86, Par87].

Johnson [GJ79] but adapted here to P-completeness.) $\square$

Observation 2.1 tells us that we may use the following much more practical approach to prove that a problem Π is P-complete.

1. Show that $\Pi \in P$.

2. Show that there exists a NC-reduction from some known P-complete problem to Π.

However we still have the "chicken before egg problem"—we simply can not use this approach to prove some first problem to be P-complete!

The first problem shown to be P-complete was the PATH problem[25] presented by Stephen Cook in 1973 [Coo74]. Shortly after, Jones and Laaser [JL77] showed six other problems to be P-complete. Cook proved the P-completeness of the PATH problem by describing a generic log-space reduction of an arbitrary problem in P to the PATH problem. Similarly, Jones and Laaser provided a generic reduction for the UNIT RESOLUTION problem[JL77].

Informally, such a generic reduction (to e.g. the PATH problem) is a description of how an arbitrary deterministic polynomial time computation on a Turing machine may be simulated by solving a corresponding instance of the PATH problem. The generic reduction to the PATH problem invented by Cook is an extremely important result in the history of (parallel) complexity theory. This, and other similar generic reductions are complicated mathematical descriptions—which we do not need to know in detail to use the P-completeness theory in practice.

In addition to the original papers ([Coo74, JL77]) interested readers are referred to Chapter 7 in [GR88] which describes a generic reduction to the GENERABILITY problem. See also the very good description of Stephen Cooks seminal theorem (*Cook's Theorem*) found in Section 2.6 of [GJ79]. This theorem provided the first NP-complete problem (SATISFIABILITY) by a generic reduction from an arbitrary problem in NP.

Part 1 of the proof procedure, to show that the problem is in P, is often a straightforward task. All we need is to show the existence of some deterministic polynomial time uni-processor algorithm solving the problem.

[25] Called PATH SYSTEM ACCESSIBILITY by Garey and Johnson [GJ79] p. 179. The problem is also described in [JL77].

One exception is "linear programming" which was shown to be in P by Khachian as late as in 1979 [Kha79].

To find a NC-reduction from some known P-complete problem is in general not so easy. A good way to start is to study documented P-completeness proofs of related problems. Chapter 3 in the book by Garey and Johnson [GJ79] contains a lot of fascinating problem transformations, and it is highly recommended—in spite of the fact that the problem transformations reported there are polynomial reductions and not necessarily NC-reductions. However, Anderson reports that most of the transformations used in NP-completeness proofs can also be performed as NC-reductions [And86].

What is difficult is to select a well suited known P-complete problem and to find a systematic way of mapping instances from that problem into instances of the problem we want to prove P-complete. Once the reduction has been found, it is often relatively easy to see that it can be performed by a NC-algorithm. That part is therefore omitted from most P-completeness proofs.

The Circuit Value Problem

Since the pioneering work by Cook in 1973 we have got an increasing number of known P-complete problems. The most frequently used problem in P-completeness proofs is the *circuit value problem* (CVP) and its variants [And86]. The circuit value problem was shown to be P-complete by Ladner in 1975 [Lad75].

Informally, an instance of the circuit value problem is a combinational circuit (*i.e.* a circuit without feedback loops) built from Boolean two-input gates and an assignment to its inputs. To solve the problem means to compute its output [Par87]. The kind of gates allowed in CVP plays an important role. CVP is P-complete provided that the gates form a so-called complete basis [Par87, And86]. The most common set of gates is {AND, OR, NOT}. Two important variants of CVP are the *monotone* circuit value problem (MCVP) where we are restricted to use only AND or OR gates and the *planar* circuit value problem (PCVP) where the circuit can be constructed (on the plane) without wire crossings. MCVP and PCVP were both proved P-complete by Goldschlager [Gol77]. Other variants of CVP are described in [GSS82, And86, Par87]. Note that small changes in a problem definition can make dramatic effects on the possibility of a fast parallel solution. As an example, both MCVP and CVP are P-complete, but the monotone *and* planar CVP can be solved by a NC-algorithm.

P-complete problems—Examples

The set of problems proved and documented to be P-complete contains a growing set of interesting problems. Below is a short list containing some of the more important of these. The reader should consult the references for more detailed descriptions of the problems and their proofs (see also [GR88]).

Maximum network flow proved P-complete by Goldschlager et. al. in 1982 [GSS82].

Linear programming proved as member of P by Khachian [Kha79] and proved as P-complete or harder[26] by Dobkin et. al. in 1979 [DLR79].

General deadlock detection proved P-complete by Spirakis in 1986 [Spi86].

Depth first search proved P-complete by John Reif in 1985 [Rei85].[27]

Unification proved P-complete by Dwork, Kanellakis and Mitchell 1984 [DKM84]. Kanellakis describes in [Kan87] how this result depends on the representation of the input.

Unit resolution for propositional formulas proved P-complete (among several other problems) by Jones and Laaser in 1977 [JL77].

Two player game proved P-complete by Jones and Laaser in 1977 [JL77].

A P-complete problem is often termed as *inherently serial*—it often contains a subproblem or a constraint which requires that any solution method must follow some sort of serial strategy. For some problems such as "two player game" (and perhaps also "general deadlock detection") it is relatively easy to intuitively see the inherent sequentiality. For others such as "maximum network flow" it is much harder.

In general it is difficult to claim that a *problem* is inherently serial because it in our context means that all possible algorithms for solving it are inherently serial. On the contrary, it is often easier to identify inherently serial *algorithms*. This is captured in the notion of P-complete algorithms discussed in the following paragraph.

[26]This is often termed *P-hard*. Informally it means that the problem may be outside P, but *if* it can be proved as member of P then P-hardness implies P-completeness.

[27]It is more correct to term depth first search as a P-complete *algorithm*, see page 43.

P-complete algorithms

A vast amount of research has been done on sequential algorithms. When designing a parallel algorithm for a specific problem it is often fruitful to look at the best known sequential algorithms. Many sequential algorithms have fairly direct parallel counterparts.

Unfortunately, many of the most successful sequential algorithms are very difficult to translate (transform) into very fast parallel algorithms. Such algorithms may be termed inherently serial. They are often using a strategy which is basing decisions on accumulated information. A good example is J. B. Kruskal's minimum spanning tree algorithm [Kru56] which is a typical greedy algorithm (see for instance [AHU82] pages 321–324). Greedy algorithms are in general sequential in nature—they typically build up a solution set item by item, and the choice of which item to add depends on the previous choices.

In his thesis *The Complexity of Parallel Algorithms* [And86] Richard Anderson shows that greedy algorithms for several problems are inherently serial—he proves them to be P-complete. Anderson gives the following definition of a P-complete algorithm [And86]:

> An algorithm A for a search problem is *P-complete* if the problem of computing the solution found by A is P-complete.

Another example of an inherently serial algorithm is depth-first search (DFS) ([Tar72]) which is used as a subalgorithm in many efficient sequential algorithms for graph problems. John Reif has shown that the DFS algorithm is P-complete by proving that the DFS-ORDER problem[28] is P-complete [Rei85].

Fortunately, the fundamental difference between problems and algorithms makes the detection of an algorithm to be P-complete to a less discouraging result. A P-complete algorithm does not say that the corresponding problem is inherently serial—it merely states that a different approach than parallelising the sequential algorithm must be attempted to possibly obtain a fast way to solve the problem with parallelism. For example, the natural greedy algorithm for solving the maximal independent set problem is P-complete, but a different approach can be used to solve the problem with a NC-algorithm [And86, GS89].

[28] Informally described, this problem is to find the depth-first search visiting order of the nodes in a directed graph.

Table 2.2: Summary of complexity theory for sequential and parallel computations.

	sequential computations	**parallel computations**
The main class of problems studied:	NP	P
The "easiest" problems in the main class, *i.e.* the class of problems that may be solved "efficiently":	The class P	The class NC
The fundamental question:	Is $P = NP$?	Is $NC = P$?
Definition of a problem that may be solved "efficiently":	The existence of a sequential algorithm solving the problem in polynomial time	The existence of a parallel algorithm solving the problem in polylogarithmic time using a polynomial number of processors
The "hardest" problems in the main class, *i.e.* problems that probably can not be solved "efficiently":	The NP-complete problems	The P-complete problems
A "first" problem in the hardest class:	SATISFIABILITY (SAT)	PATH
Technique for proving membership in the "hardest" class:	P-reducibility	NC-reducibility

NP-completeness vs. P-completeness
Table 2.2 shows the similarities of *P*-completeness and *NP*-completeness. It does also give a quick summary of the theory.

2.2 Evaluating Parallel Algorithms

2.2.1 Basic Perfomance Metrics

In the rest of this thesis it is assumed that algorithms are executed on the CREW PRAM model. This assumption makes it possible to define the basic performance metrics for evaluating parallel algorithms.

Time, Processors and Space

Definition 2.14 (Time)
The time, running time, or execution time for an algorithm[29] executed on a CREW PRAM is the number of CREW PRAM clock periods elapsed from the first to the last CREW PRAM instruction used to solve the given instance of a problem. The time is expressed in CREW PRAM time units or simply time units. □

This definition corresponds to what many authors call parallel (running) time [Akl85]. A large number of processors may perform the same or different CREW PRAM instructions in one CREW PRAM clock period[30]. Definition 2.14 makes it possible to use the same concept of time for parallel algorithms and serial algorithms executed as uni-processor CREW PRAM programs. Note that the definition implies that time used by an algorithm strongly depends on the size and possibly also the characteristics of the actual problem instance.

On models which reflect parallel architectures based on message passing it is common to differentiate between computational steps and routing steps (see for instance [Akl85, Akl89]). As described in Chapter 3, access to the global memory from the processors in a CREW PRAM is done by one single

[29]More precisely, a CREW PRAM *implementation* of an algorithm.

[30]Each processor is assumed to have a rather simple and small instruction set. (Nearly all instructions use one time unit, some few (such as divide) use more. The instruction set time requirement is defined as parameters in the simulator—and therefore easy to change. See Appendix A.)

instruction, and it is therefore less important to differentiate between local computations and inter-processor communication.

Definition 2.15 (Processor requirement)
The processor requirement of an algorithm is the maximum number of CREW PRAM processors that are active in the same clock period during execution of the algorithm for a given problem instance. □

The processor requirement is central in the evaluation of parallel algorithms since the number of processors used is a good representative of the "hardware cost" which is needed to do the computations.

Definition 2.16 (Space)
The space used by a CREW PRAM algorithm is the maximum number of memory locations in the global memory allocated at the same time during execution of the algorithm for solving a given problem instance. □

Note that we do not measure the space used in the local memories of the CREW PRAM processors. This is because it is easy to attach cheap memories to the local processors, while it is difficult and expensive to realise the global memory which is accessible by all the processors. Little emphasis is put on analysing the space requirements of the algorithms evaluated in this thesis.

Cost, Optimal Algorithms, and Efficiency
It is often possible to obtain faster parallel algorithms by employing more processors. Many of the fastest algorithms require a huge number of processors—often far beyond what can be realised with present technology. It is therefore frequently the case that the fastest algorithm is not the "best" algorithm.

One way to get the question of feasibility into the analysis is to measure or estimate the cost of the parallel algorithm.

Definition 2.17 (Cost) *([Akl89] p. 26, [Qui87] p. 43)*
The cost of a parallel algorithm is the product of the time (Definition 2.14) and the processor requirement (Definition 2.15). The cost is expressed in number of CREW PRAM (unit-time) instructions. □

Note that this definition "assumes" that all the processors are active during the whole computation. This is a simplification which makes the cost an upper bound of the total number of CREW PRAM instructions performed. In contrast to Akl [Akl85, Akl89] which define cost to represent worst case behaviour, we define cost to be dependent on the actual problem instance (see Definitions 2.14 and 2.15).

There are many ways to define a parallel algorithm to be *optimal* for a given problem, depending on what resources are considered most critical. It is most common to say that a parallel algorithm is optimal (or cost optimal) if its cost (for a given problem size) is the same as the cost of the best known sequential algorithm (for the same problem size)—which is known to be $O(n \log n)$ [Akl85, Knu73].

Definition 2.18 (Optimal parallel sorting algorithm)
A parallel sorting algorithm is said to be optimal (with respect to cost) if its cost is $O(n \log n)$. $\square$

The fastest parallel algorithms often use redundancy, and are therefore not optimal with respect to cost. It is often desirable with a figure which tells how good the utilisation of the processors is. This is achieved by the measure *efficiency*.

Definition 2.19 (Efficiency)
The efficiency of a parallel algorithm is the running time of the fastest known sequential algorithm for solving a specific instance of a problem divided by the cost of the parallel algorithm solving the same problem. $\square$

Efficiency may also be defined as the *speedup* divided by the processor requirement, see Section 2.2.2. For a discussion of the possibility of efficiency greater than one, see Section 2.2.2.2.

The term *efficient* is used in connection with parallel algorithms in many ways. Gibbons and Rytter [GR88] defines an efficient parallel algorithm to be an algorithm with polylogarithmic time consumption using a polynomial number of processors. According to this, an algorithm with an extensive use of processors may be termed as efficient even though it may have a very low efficiency.

Richard Karp and others [Kar89] have advocated an alternative meaning of *efficient parallel algorithm* which does not allow parallel algorithms that is "grossly wasteful" of processors to be termed as efficient. They define informally a parallel algorithm to be efficient if its cost is within a polylogarithmic factor of the cost (*i.e.* execution time) of the best sequential algorithm for the problem.

2.2.2 Analysis of Speedup

2.2.2.1 What is Speedup?

The most frequently used measure of the effect that can be obtained by parallelisation is speedup. The speedup does express how much faster a problem can be solved in parallel than on a single processor. Many different definitions of speedup can be found in the literature on parallel processing. A precise definition of the most common use of speedup is given below.

Definition 2.20 (Speedup)
Let $T_1(\Pi)$ be the execution time of the fastest known sequential algorithm for a given problem Π run on a single processor of a parallel machine M.[31] Let $T_N(\Xi)$ be the execution time of a parallel algorithm Ξ for the problem Π run on the same parallel machine M using N processors. The **speedup** achieved by the parallel algorithm Ξ for the problem Π run on N processors is defined as

$$Speedup(\Pi, \Xi, N) = T_1(\Pi)/T_N(\Xi) \tag{2.1}$$

□

Some variants of this definition should be mentioned. The definition given by Akl in [Akl85] specifies that both the sequential and parallel execution times are *worst case execution times.*

Miklośko in [MK84] requires that the sequential algorithm must be the fastest *possible* algorithm for a single processor, which often is unknown. Comparing with the fastest known algorithm is more practical.

Quinn in [Qui87] emphasises the use of a single processor of the parallel machine for execution of the sequential algorithm. The alternative is to use the *fastest known serial computer* which may be regarded as giving a more "correct" figure of the speedup. However, few researchers (in parallel processing) have access to the most powerful serial computers.

One popular alternative exists that often gives better speedup results than Definition 2.20. In this alternative definition, the sequential running time is measured or estimated as the time used by the *parallel* algorithm run on a *single* processor. The main error introduced is that the redundancy often existing in good parallel algorithms in this case also must be performed in the uni-processor solution, giving a poorer sequential running time. The main motivation for this strategy is that it does reduce the research to a

[31] It is implicitly assumed that all processors on this machine are of equal power.

study of *only* the parallel algorithm. One example of this alternative speedup definition can be found in [Moh83].

2.2.2.2 Superlinear Speedup—Is It Possible?

The term linear speedup often appears in the literature, and there is a discussion whether so-called superlinear speedup is possible or not. Quinn [Qui87] defines *linear speedup* to be a speedup function, $S(N)$, which is $\Theta(N)$ (of order *exactly* N), where N is the number of processors. Thus $\frac{1}{2}N$ and $2N$ both express linear speedup according to this definition. Some other authors only regard $S(N) = N$ as linear speedup.

Superlinear speedup is used to denote the case (if it exists) where a problem is solved more than N times faster using N processors—*i.e.* a speedup greater than linear. Note that the chosen definition of linear speedup is crucial for such a discussion. Parkinson in [Par86] gives a very simple example with speedup $S(N) = 2N$. According to the chosen definition of linear speedup, this example is, or is not, exhibiting superlinear speedup. (See also [HM89]).

2.2.3 Amdahl's Law and Problem Scaling

This section starts with a description of the reasoning behind Amdahl's law, and it is shown that a recently announced alternative law is strongly related to Amdahl's law. This discussion leads to the concept of problem scaling. The theory is then illustrated by a simple example. The last part of the section shows why it is difficult to use Amdahl's law on practical algorithms. We see that the law is only useful as a coarse model for giving an upper bound on speedup. The discussion reveals various issues which should be covered in a more detailed analysis method.

2.2.3.1 Background

In 1967, Gene Amdahl in [Amd67] gave a strong argument against the use of parallel processing to increase computer performance. Today, this short article seems rather pessimistic with respect to parallelism, and his claims have become *partly* obsolete.

However, some of his predictions have shown to be right in nearly two decades, and the reasoning is considered as the origin of a general speedup formula called *Amdahl's law*.

In his article, Amdahl first describes that practical computations have a significant sequential part[32]. The effect of increasing the number of processors to speed up the computation will soon be limited by this more and more dominating sequential part. By using Grosch's law[33] he then argues that more performance for the same cost is achieved by upgrading a sequential processor than by using multiprocessing. Amdahl further describes that it has always been difficult to exploit additional hardware in improving a sequential computer, but that the problems have been overcome. He predicts that this will continue to happen in the future.

In short, Amdahl considers a given computation having a fixed amount of operations that must be performed sequentially and how the computation may be sped up by using several processors. This is called *Amdahl reasoning* in the following.[34] If the sequential fraction of the computation is called s, by Amdahl's law the maximum speedup that can be achieved by parallel processing is bounded by $1/s$. This observation has often been used as a strong argument against the viability of *massive parallelism*[35].

The implicit assumption underlying Amdahl's law has been criticised as inappropriate in the past. An alternative "inverse Amdahl's law" has been suggested by a research team from Sandia National Laboratories, [Gus88, GMB88, San88]. They have achieved very good speedup results for three practical scientific applications. They use an inverted form of Amdahl's argument to develop a new speedup formula for explaining their results. This new formula is also claimed to show that it in general is much easier to achieve efficient parallel performance than implied by Amdahl's paradigm.

In my opinion, this new formula, just as Amdahl's law, can be derived from a very standard speedup calculation. As will be shown, the two formulas are strongly related. Nevertheless, *the new law looks much more optimistic with respect to the viability of parallelism. The reason for this is*

[32]Amdahl found that about 35 % of the operations are forced to be sequential, 25 % is due to "data management housekeeping", and 10 % to "problem irregularities".

[33]Grosch's law states that the speed of computers is proportional to the square of their cost. It is discussed at page 18 in [Qui87].

[34]Amdahl assumed that the *sequential part* (and therefore also the parallel part) of the total computation both are *unaffected by the number of processors used* in the parallel execution. In retrospect, I think this implicit assumption is the crucial part of the article, and the reason why the law has been named after Amdahl. Several authors [Qui87, Gus88] refer to [Amd67] as the source of Amdahl's law. However, the arguments of Amdahl are rather informal and are formulated as a belief (prophecy). No explicit law is defined.

[35][GMB88] uses the term massive parallelism for general purpose MIMD–systems with 1000 or more autonomous floating point processors.

an implicit assumption which is hidden by the derivation of the law. Furthermore, standard speedup analysis gives the same more optimistic picture as the new formula, provided that the same assumptions are made. In this light I think it is appropriate to discuss in more detail Amdahl's law and the alternative speedup formula suggested by the team from Sandia.

2.2.3.2 Amdahl's Law

Definition 2.21 (Serial work, parallel work)
Consider a given algorithm Λ. Let $T_1(\Lambda)$ be the total time used for executing Λ on a single processor. The part of the time $T_1(\Lambda)$ used on computations which *must* be performed sequentially[36] is denoted s_w. The remaining part of the time $T_1(\Lambda)$ is used on computations that *may* be done in parallel, and is denoted p_w. s_w is called the *serial part* (of the uni-processor execution time). It is sometimes called the *serial work.* Correspondingly, p_w is called the *parallel part*, or *parallel work.* By definition, $s_w + p_w = T_1(\Lambda)$. $\square$

Definition 2.22 (Serial fraction, parallel fraction)
Let s denote the *serial fraction* of the uni-processor execution time, and p the *parallel fraction* of the uni-processor execution time. s and p are given by:

$$s = \frac{s_w}{s_w + p_w} \quad , \quad p = \frac{p_w}{s_w + p_w} \tag{2.2}$$

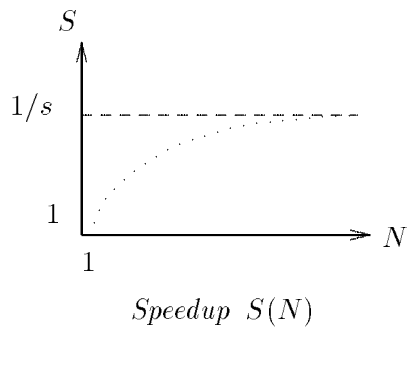
$\square$

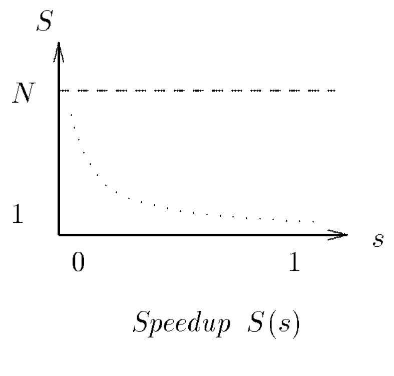
Now consider executing the same algorithm Λ on a parallel machine with N processors, each with the same capabilities as the single processor used above. The time used is $T_N(\Lambda)$. Assume that the parallel work p_w can be split evenly among the N processors without any extra overhead. The execution time by this *direct parallelisation* of Λ is then $T_N(\Lambda) = s_w + p_w/N$. The speedup achieved is given by

$$S_{Amdahl} = \frac{T_1(\Lambda)}{T_N(\Lambda)} = \frac{s_w + p_w}{s_w + p_w/N} = \frac{1}{s + p/N} = \frac{1}{s + (1-s)/N} \quad , \tag{2.3}$$

has often been used to plot the possible speedup as a function of N (for fixed s), or as a function of s (for fixed N), see Figure 2.4.

[36] Sequentially here means that this part of the computation must be performed step by step (i.e. sequentially), and *alone.* No other parts of the total computation may be executed while the serial part is performed.





a) b)

Figure 2.4: Two interpretations of S_{Amdahl}.

2.2.3.3 "Inverted" Amdahl's Law

Amdahl's law can be viewed as a result of considering the execution time of a given serial program run on a parallel machine, [GMB88]. The team at Sandia has derived a new speedup formula, here called S_{Sandia}, by inverting this reasoning: They consider how much time is needed to execute a given parallel program on a serial processor.

Definition 2.23
Consider a given parallel algorithm Ξ. Let $T_N(\Xi)$ be the total time used for executing Ξ on a parallel machine using N processors. The part of the time $T_N(\Xi)$ used on computations which *must* be performed sequentially is denoted s'_w. The remaining part of the time $T_N(\Xi)$ is used on computations that *may* be done in parallel, and is denoted p'_w. s'_w is called the *serial part* (of the N-processor execution time). Correspondingly, p'_w is called the *parallel part* (of the N-processor execution time). By definition, $s'_w + p'_w = T_N(\Xi)$. $\Box$

Definition 2.24
Let s' denote the *serial fraction* of the N-processor execution time, and p' the *parallel fraction* of the N-processor execution time. s' and p' are given

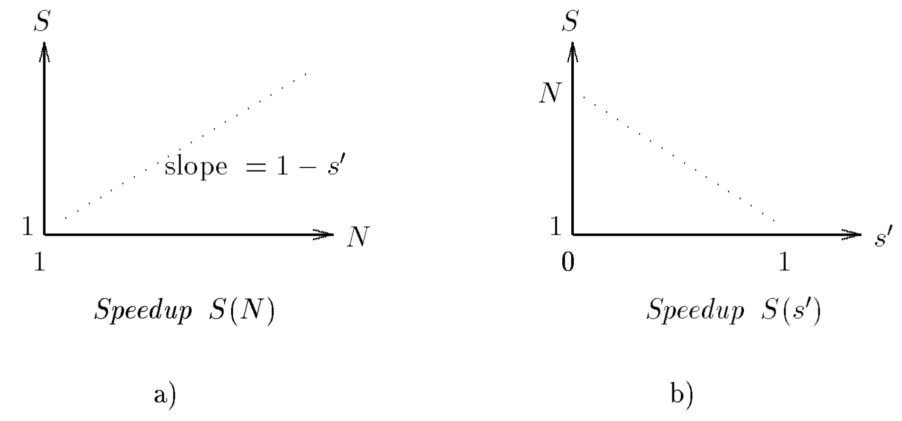


Figure 2.5: Two interpretations of S_{Sandia}.

by:

$$s' = \frac{s'_w}{s'_w + p'_w} \quad , \quad p' = \frac{p'_w}{s'_w + p'_w} \tag{2.4}$$

□

On a single processor, the time used to execute Ξ, $T_1(\Xi)$ is $s'_w + p'_w N$. This gives the following alternative speedup formula:

$$S_{Sandia} = \frac{T_1(\Xi)}{T_N(\Xi)} = \frac{s'_w + p'_w N}{s'_w + p'_w} = s' + p'N = s' + (1 - s')N = N + (1 - N)s' \tag{2.5}$$

S_{Sandia} is shown as a function of N (for fixed s'), and as a function of s' (for fixed N) in Figure 2.5.

Discussion

At first sight, it seems like this inverted Amdahl's reasoning gives a new and much more optimistic law. S_{Sandia} expresses that speedup increases linearly with N for a fixed s'. The upper bound for possible speedup expressed by Amdahl's law has been removed. We may be led to believe that Amdahl's law is without practical value when we consider uni–processor execution of a parallel computation, or even worse — that the law is wrong.

It is important here to realise the effects of keeping s fixed instead of keeping s' fixed when N is varied. This is in fact the only source for the difference between the two formulas[37]: In Amdahl's reasoning, where we have a fixed uni-processor execution time, keeping s *fixed* while N increases corresponds to computing a fixed amount of parallel work p_w faster and faster by using more processors. However, in the inverted reasoning which assumes fixed N-processor execution time, keeping s' *fixed* while N increases corresponds to performing a larger and larger parallel work[38] in fixed time p'_w at the parallel system, while the serial work remains unchanged.

These two situations are rather different. Keeping s' fixed when N increases corresponds to decreasing s and thus "lifting the upper bound"[39] in part a) of Figure 2.4. Similarly, keeping s fixed with varying N corresponds to increasing s' and thus reducing the slope of the curve in part a) of Figure 2.5.

2.2.3.4 Problem Scaling and Scaled Speedup

Discussion

Which speedup formula is correct? Both are. The team at Sandia has announced their formula because they think it better describes parallelisation efforts as it is done in practice. Gustafson in [Gus88] argues that Amdahl's law is most appropriate for academic research:

> One does not take a fixed-sized problem and run it on various numbers of processors except when doing academic research; in practice, *the problem size scales with the number of processors.*

Agreeing with this statement or not, there certainly exist situations where it is natural to scale the problem with the number of processors available. Often, the fixed quantity is not the problem size but rather the amount of time a user is willing to wait for an answer.[40] In handling three real problems, wave mechanics, fluid dynamics and beam strain analysis, the team

[37]Both for the direct parallelisation leading to S_{Amdahl} and the direct serialisation giving S_{Sandia} we have in general that $s_w = s'_w$ and $p_w = p'_w N$. If we in Equation 2.3 substitute s_w and p_w with s'_w and $p'_w N$ respectively, we get the new speedup formula given by Equation 2.5. Similarly S_{Amdahl} can be directly derived from S_{Sandia}.

[38]This *work* is the time needed to execute the parallel computation on a single processor, i.e. $p_w = p'_w N$. Thus, the parallel work increase linearly in N.

[39]This is seen directly from the definition of s when s_w is fixed and p_w increases.

[40]A good example is *weather forecasting* which illustrates both the fixed-sized model (Amdahl reasoning) and the scaled-sized model (inverted Amdahl reasoning). Assume that the weather prediction is guided by a simulation model of the atmosphere. Two important

at Sandia found it natural to increase the size of the problem when using more processors. This is what they call *problem scaling.*

In their work, they found that it was the parallel part of the program that scales with the problem size. Therefore, as an approximation, the amount of work p_w, that can be done in parallel varies linearly with the number of processors N. In other words, since $p'_w = p_w/N$ and p_w varies linearly with N, such problem scaling corresponds to keeping s' fixed. Therefore, under these conditions Figure 2.5 gives a correct picture of the possible speedup. The team at Sandia called this speedup measure for *scaled speedup* to reflect that it assumes problem scaling.

(This section was written in December 1988. Similar thoughts have later been expressed by several authors, see for instance the discussion in [HWG89] and [ZG89], and the detailed treatment given in [SN90].)

2.2.3.5 A Simple Example, Floyd's Algorithm

The intention of the following example is to illustrate parts of the theory described above. It also gives a background for a further discussion of Amdahl's law in Section 2.2.3.6.

The famous Floyd – Warshall algorithm[41] for the *all pairs* shortest path problem is shown in Figure 2.6. The reader is referred to pages 104–107 of [CM88] for further details about the algorithm. The variable names in Figure 2.6 are the same as used in [CM88]. Other descriptions of the algorithm can be found in pages 208–211 of [AHU82], and pages 86–90 of [Law76].

The execution of this algorithm on a single processor requires $O(n^3)$ time. There are exactly n^3 instances of the assignment in line (4). [CM88] shows that the n^2 *assignments* given by line (4) for a fixed value of k in the outermost for-loop *are independent.* Therefore, they can be executed in any order, or simultaneously (*i.e.* in parallel). Thus the algorithm may

factors for the quality of the forecasting are 1) how new the inputs (weather samples) to the simulation model are, and 2) how detailed the simulation is. If the computer used for the simulation is upgraded by increasing the number of processors, there are principally two main alternatives for quality improvement of the forecasting; *Amdahl reasoning:* Solve the same problem faster, i.e. use newer weather data in the simulation. *Inverted Amdahl reasoning:* Solve a bigger problem within the same execution time, i.e. use a more detailed simulation model.

[41]The names *Floyd's algorithm* and *the Warshall-Floyd algorithm* do also appear in the literature. The algorithm was first presented by Robert W. Floyd in [Flo62], and is based on a theorem on Boolean matrices published by Stephen Warshall in [War62].

```
    SEQUENTIAL procedure Floyd;
    { Initially all $d[i,j] = W[i,j]$ }
    begin
(1)     for $k := 1$ to $n$ do
(2)         for $i := 1$ to $n$ do
(3)             for $j := 1$ to $n$ do
(4)                 $d[i,j] := \min(d[i,j], d[i,k] + d[k,j])$
    end;
```

Figure 2.6: Floyd's algorithm.

be executed in $O(n)$ time on a synchronous machine with $O(n^2)$ processors. (See program P4 in Section 5.4.1. in [CM88].)

We will now use the theory presented in this section on the parallelisation described above for the algorithm *Floyd*. Assume that the "loop control" for one iteration of a for loop takes one time unit, and that one execution of line (4) requires four time units. The serial work s_w of the computation consists of line (1) taking n time units. The parallel work p_w consists of line (2), (3) and (4) taking n^2, n^3, $4n^3$ time units respectively. $T_1(\mathit{Floyd}) = s_w + p_w = 5n^3 + n^2 + n$. Assuming that the parallel part of the computation can be distributed evenly among N processors without extra overhead, we have $T_N(\mathit{Floyd}) = s_w + p_w/N = n + (5n^3 + n^2)/N$.

Now consider a fixed problem size, for instance $n = 100$. The serial fraction of the uni-processor execution time is $s = n/(5n^3 + n^2 + n) = 1/50101 \approx 0.002\%$. Hence, we observe that this algorithm has a very small serial part even for relatively small problem sizes, and that $\lim_{n\to\infty} s = 0$. The speedup as a function of the number of processors used (N) is then given by Amdahl's law in Equation 2.3,

$$S_{Amdahl} = \frac{1}{s + (1-s)/N} = \frac{50101 \times N}{N + 50100} \tag{2.6}$$

This corresponds to the general speedup curve shown in part a) of Figure 2.4. For $N \ll 50101$ Equation 2.6 does express a *"nearly linear speedup"*, and for $N = n^2 = 10000$ we get $S \approx (5/6)N$. Further we see that the maximum speedup is limited by $1/s = 50101$ using an infinitely large number of processors for this fixed sized problem.

Let us now consider the speedup formula derived by the team at Sandia.

We have $s'_w = n$, and the time used to execute the parallel part of the computation using N processors is $p'_w = p_w/N = (5n^3 + n^2)/N$. Further, the serial fraction of the *parallel execution time* is

$$s' = n/[n + (5n^3 + n^2)/N] \tag{2.7}$$

When we used Amdahl's law above we assumed a fixed problem size n — corresponding to a fixed value for the serial fraction s of the uni-processor execution time. In *this* case we assume that s' is kept fixed, which requires that the problem size is scaled with the number of processors. The necessary relation between n and N can be found by solving Equation 2.7 with respect to n or N, for instance $N = (5n^2 + n)s'/(1 - s')$. Now, assume that this relation holds between N and n, and that $s' = \frac{1}{2}$. Using the Sandia speedup formula in Equation 2.5 we get the speedup as a function of N

$$S_{Sandia} = s' + (1 - s')N = \frac{1}{2}(5n^2 + n + 1) \tag{2.8}$$

For large n, where $n^2 \gg n$ keeping s' fixed at $\frac{1}{2}$ corresponds to scaling the problem size by setting $n = \sqrt{N/5} \approx 0.45\sqrt{N}$. In this case Equation 2.8 gives $S \approx \frac{1}{2}N$, which corresponds to part a) of Figure 2.5.

This example shows that a rather moderate problem scaling gives an *"unlimited"* speedup function. The upper bound for the speedup given by Amdahl's law occurs because only a limited number of processors can be used effectively on a *fixed amount* of parallel work. This upper bound is removed if we increase the amount of parallel work in correspondence with the number of processors used. It should therefore be no surprise that it is easier to obtain optimistic speedup results if we allow problem scaling.

2.2.3.6 Using Amdahl's Law in Practice

There are many problems associated with the use of Amdahl's law in practice. This section aims at illuminating some of the problems, not solving them.

Choice of parallelisation

Consider an arbitrary given sequential program, and the question "What speedup can be achieved by parallel processing this program?". The first issue encountered is the *choice of parallelisation.* In this context, an optimal parallelisation may be defined as one which gives the "best" speedup curve.

But what is the "best" speedup curve? Is it most important to obtain a high speedup for reasonably low values of N (ten's or hundred's of processors), or should one strive for a parallelisation giving the highest asymptotic ($N \to \infty$) speedup value?

Even if one could decide upon a definition of an optimal speedup curve, we still have the generally difficult problem of finding the parallelisation giving this speedup, and proving that it is optimal. However, there exists a large number of non–optimal parallelisations for most problems, and most of them are interesting in some manner. In the literature the most usual situation is *speedup analysis of a given parallelisation*, which is the approach adopted here.

Calculation of serial and parallel work

Assume a given parallelisation of a sequential program. In the general case, identifying the amount of serial and parallel work will be a more complicated task than examplified by the analysis of Floyd's algorithm in Section 2.2.3.5. The calculation of s_w and p_w for the described parallelisation of *Floyd* was straightforward partly due to the simplicity of the algorithm, and partly due to some implicit and simplifying assumptions.

In Figure 2.6 we assumed that all the work in lines (4) could be done in parallel. The amount of parallel work for line (4) was calculated by accumulating the effect of the three enclosing for–loops. Now suppose that each single execution of line (4) has to be performed serially. What is now the amount of serial work described by line (4)? If we use the *"accumulated amount of work"* for line (4) which is $4n^3$, we get a serial work $s_w = 4n^3 + n$. If this is used together with the corresponding $p_w = n^3 + n^2$, Amdahl's law gives a *constant* speedup of approximately $\frac{5}{4}$ for large values of n and $N = n^2$. However, using n^2 processors makes *Floyd* into a simple loop with n iterations, each of constant time. Thus $N = n^2$ processors gives a speedup of order $O(n^2)$. Therefore, *the ad hoc method for calculating s_w and p_w used in Section 2.2.3.5 can not be used in the general case.*

These problems are caused by a mix of serial and parallel parts in the *nested* program structure. Branching and procedures will certainly introduce further difficulties in the calculation of s_w and p_w.

Another question which arise is to what extent it is correct to calculate one global measure of the parallel work p_w as the sum of the parallel work in various parts of the program.

How many processors can effectively be exploited ?

Now assume that the parallel work p_w has been calculated and that "overlapping of all parallel work" is possible. According to the theory, in this case $N = p_w$ number of processors may be used to execute the parallel work in one time unit. This is in most practical cases very unrealistic. Distributing *all* the parallel work *evenly* among the N processors may be impossible. A more detailed analysis which takes into account the varying number of processors which may be used in the execution of the various parts of the parallel work is needed.

Chapter 3

The CREW PRAM Model

This chapter is devoted to the CREW PRAM model and how it may be programmed. Section 3.1 starts by describing the model as it was proposed for use in theoretical computer science by Fortune and Wyllie [FW78, Wyl79]. This description is then extended by some few details and assumptions that are necessary to make the model to a vehicle for implementation and execution of algorithms. The following two sections describe how the CREW PRAM model may be programmed, and a high level notation for expressing CREW PRAM programs is outlined. Section 3.4 ends the chapter by describing a top down approach that has shown to be useful for developing parallel programs on the CREW PRAM simulator system. The approach is described by an example.

3.1 The Computational Model

> "Large parallel computers are also difficult to program. The situation becomes intolerable if the programmer must explicitly manage communication between processors."
>
> A. G. Ranade, S. N. Bhatt and S. L. Johnsson in *The Fluent Abstract Machine* [RBJ88].

> "The reason is that most all parallel algorithms are described in terms of the PRAM abstraction, which is a practice that is not likely to change in the near future."
>
> Tom Leighton in *What is the Right Model for Designing Parallel Algorithms?* [Lei89]

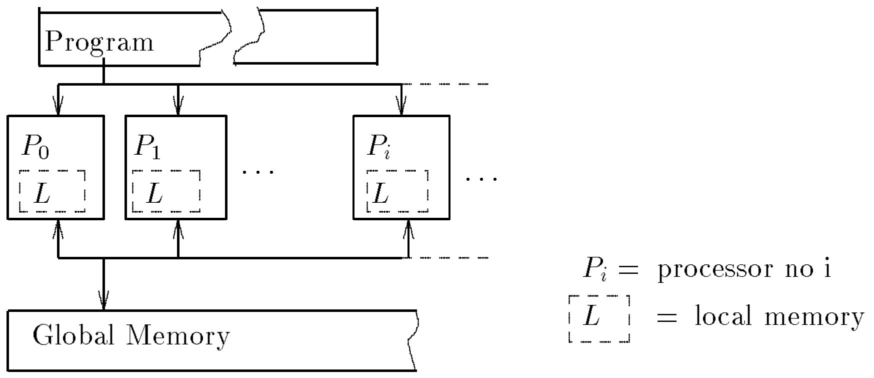


Figure 3.1: The CREW PRAM model.

"Based on quite a few years of experience in designing parallel algorithms, the author believes that difficulties in designing parallel algorithms, will make any parallel computer that does not support the PRAM model of parallel computation less than competitive (as a general purpose computer) with the strongest non parallel computer that will become available at the same time."

Uzi Vishkin in *PRAM Algorithms: Teach and Preach* [Vis89].

In this section we first describe the original CREW PRAM model as it was proposed by Fortune and Wyllie [FW78, Wyl79]. This is followed by a documentation of some additional properties that have been necessary to define for making the CREW PRAM model into a vehicle for executing and measuring parallel algorithms. (The main characteristics of the CREW PRAM have been collected in small "definitions" with the heading "**CREW PRAM property**". This have been done for easy reference.)

3.1.1 The Main Properties of The Original P-RAM

A CREW PRAM is a very simple and general model of a parallel computer as outlined in Figure 3.1. It consists of an unbounded number of processors

Instruction		Function
LOAD	operand	Transfer operand to/from the
STORE	operand	accumulator from/to memory.
ADD	operand	Add/proper subtract the value of
SUB	operand	the operand to/from the accumulator.
JUMP	label	Unconditional branch to label.
JZERO	label	Branch to label if accumulator is zero.
READ	operand	See text.
FORK	label	See text.
HALT		See text.

Figure 3.2: CREW PRAM instruction set. (From [Wyl79]).

which are connected to a global memory of unbounded size. The processors are controlled by a single, finite program.

Processors

A CREW PRAM has an unbounded number of equal processors. Each has an unbounded local memory, an accumulator, program counter, and a flag indicating whether the processor is running or not.

Instruction set

In the original description of the P-RAM model the instruction set for the processors was informally outlined as shown in Figure 3.2. This very small and simple instruction set was sufficiently detailed for Fortune and Wyllie's use of the P-RAM as a theoretical model. Each operand may be a literal, an address, or an indirect address. Each processor may access either global memory or its local memory, but not the local memory of any other processor. Indirect addressing may be through one memory to access another. There are three instructions that need further explanation;

`READ` Reads the contents of the input register specified by the operand, and places that value in the accumulator of the processor executing the instruction.

`FORK` If this instruction is executed by processor P_i, P_i selects an inactive processor P_j, clears P_j's local memory, copies P_i's accumulator into P_j's accumulator, and starts P_j running at the label which is given as part of the instruction.

`HALT` This instruction causes a processor to stop running.

Global memory
A CREW PRAM has an unbounded global memory shared by all processors. Each memory location may hold an arbitrary large non-negative integer.

CREW PRAM property 1 *(Concurrent read exclusive write)*
Simultaneous reads of a location in global memory are allowed, but if two (or more) processors try to write into the same global memory location simultaneously, the CREW PRAM immediately halts. ([Wyl79] p. 11) •

CREW PRAM property 2 *(Reads before writes)*
Several processors may read a location while one processor writes into it; all reads are completed before the value of the location is changed. ([Wyl79] p. 11) •

CREW PRAM property 3 *(Simultaneous reads and writes)*
An unbounded number of processors may write into global memory as long as they write to different locations. An unbounded number of processors may read any location at any time. (Implicit in [Wyl79]) •

Local memory
Each processor has an unbounded local memory. Each memory location may hold an arbitrary large non-negative integer.

Input and output
The input and output features of the original P-RAM reflects its intended use in parallel complexity theory. The input to a CREW PRAM is placed in the *input registers*, one bit per register.[1] When used as an acceptor, the CREW PRAM delivers its output (1 or 0) in the accumulator of processor P_0. When used as a transducer, the output is delivered in a designated area in the global memory.

[1]In some contexts, Wyllie permitted the input registers to hold integers as large as the size of the input object.

One finite program
All processors execute the same program. The program is defined as a finite sequence of labelled instructions from the instruction set shown in Figure 3.2. Fortune and Wyllie allowed several occurrences of the same label—resulting in nondeterministic programs.

Operation
The CREW PRAM starts with all memory cleared, the input placed in the input registers, and with the first processor, P_0, storing the length of the input in its accumulator. The execution is started by letting P_0 execute the first instruction of the program.

CREW PRAM property 4 *(Synchronous operation)*
At each step in the computation, each running processor executes the instruction given by its program counter in one unit of time, then advances its program counter by one unless the instruction causes a jump. ([Wyl79] p. 10) •

CREW PRAM property 5 *(Processor requirement)*
The number of processors required to accept or transform an input (*i.e.* to do a computation) is the maximum number of processors that were active (started by a `FORK` and not yet stopped by a `HALT` instruction) at any instant of time during the P-RAM's computation. ([Wyl79] p. 11) •

3.1.2 From Model to Machine—Additional Properties

3.1.2.1 Difficulties Encountered When "Implementing" Algorithms on a Computational Model

The original work by Fortune and Wyllie does not give descriptions of the P-RAM model which are detailed enough for our purpose—which is to simulate real parallel algorithms as if they were executed on a "materialised" P-RAM. The simplicity of computational models is achieved by adopting technically unrealistic assumptions, and by hiding a lot of "uninteresting" details.

Hidden details are typically how various aspects of the machine (model) should be realised. If it is commonly accepted that a certain function f may be realised within certain limits that do *not* affect the kind of analysis that is performed with the model—the details of how f is realised may be omitted. For instance, if the model is used for deriving order-expressions for the running time of programs, the detailed realisation of the function f is uninteresting, as long as it is known that it may be done in constant time. It is therefore no surprise that it is difficult to find detailed descriptions for how the CREW PRAM operates.

Nevertheless, some implicit help may be found in Wyllie's thesis [Wyl79]. In addition to its contributions to complexity theory, it describes well how the P-RAM may be *programmed* to solve various example problems. Although these descriptions are at a relatively high level, they indicate quite well *how the P-RAM was intended to be used* for executing parallel algorithms.

Implementing parallel algorithms as complete and measurable programs, made it necessary to make a lot of detailed implementation decisions. These decisions required modifications and extensions of the *original* CREW PRAM properties, defined in Section 3.1.1. They are intended to be a *"common sense" extension of Wyllie's CREW PRAM philosophy*, and are presented in the following.

3.1.2.2 Properties of The Simulated CREW PRAM

In the current version of the simulator the time consumption must (with a few exceptions) be specified by the programmer for the various parts of the program, see Section 3.4. In a more complete system, this should be done automatically by a compiler.

Extended instruction set

CREW PRAM property 6 *(Extended instruction set)*
The original instruction set shown in Figure 3.2 is extended to a more complete, but still simple instruction set. Nearly all instructions use one time unit, some few (such as divide) use more. The time consumption of each instruction is defined as parameters in the simulator—and it is therefore easy to change. •

Input and output
Our use of the CREW PRAM model makes it natural to read input from the global memory and to deliver output to global memory.

Operation
Nondeterministic programs are *not* allowed in the simulator.

CREW PRAM property 7 *(Cost of Processor Allocation)*
Allocation of n processors on a CREW PRAM initiated by one (other) processor can be done by these $n + 1$ processors in at most $C_1 + C_2\lfloor\log n\rfloor$ time units. •

In the current version of the simulator we have $C_1 = 45$ and $C_2 = 23$ time units for $n \geq 3$. These values have been obtained by implementing an asynchronous CREW PRAM algorithm for processor allocation. The algorithm uses the `FORK` instruction in a binary tree structured "chain-reaction".

Processor set and number
When programming parallel algorithms it is often very convenient to have direct access to the processor number of each processor. This feature was not included in the original description of the P-RAM. In a P-RAM used for developing and evaluating parallel algorithms it seems natural to include such a feature by letting each processor have a local variable which holds the processor number.

There are also cases where the processors need to know the address in global memory of its *processor set* data structure.[2] (The processor set concept is introduced in Section 3.2.2. See also Section A.2.3.4.) It is natural to assign this information to each processor when it is allocated.

CREW PRAM property 8 *(Processor set and processor number)*
After allocation, each processor knows the processor set it currently is member of, and its logical number in this set (numbered 1,2, ...). The processor number is stored in a local variable called p. •

The use of the simulator and related tools is documented in Appendix A.2.

3.2 CREW PRAM Programming

This section describes how the CREW PRAM model may be programmed in a high level synchronous MIMD programming style. The style follows the original thoughts on PRAM programming described by James Wyllie in his

[2]One such case is the resynchronisation algorithm outlined in Section 3.2.5.1.

thesis [Wyl79]. We present a notation called Parallel Pseudo Pascal (PPP) for expressing CREW PRAM programs. The section starts with a description of some other opinions on the PRAM model and its programming—they are included to indicated that the "state-of-the-art" is far from consistent and well-defined.

3.2.1 Common Misconceptions about PRAM Programming

The PRAM model is SIMD

A common misconception about the (CREW) PRAM model is that it is a SIMD model according to Flynn's taxonomy [Fly66] (see page 21). In a recent book on parallel algorithms [Akl89] the author writes at page 7: "... shared-memory SIMD computers. This class is also known in the literature as the Parallel Random Access Machine (PRAM) model." It *may* be that a large part of the algorithms presented for the (CREW) PRAM model is SIMD style. Michael Quinn [Qui87], and Gibbons and Rytter [GR88] do also classify the PRAM model as SIMD.

Using the PRAM as a SIMD model is a choice made by the programmers, not a limitation enforced by the model. The CREW PRAM model implies a *single program*, which does *not* necessarily imply a single instruction *stream*. The crucial point here is the local program counters as described in Wyllie's Ph.D. thesis:

> *[Wyl79] page 17: "... each processor of a P-RAM has its own program counter and may thus execute a portion of the program different from that being executed by other processors. In the notation of [Fly66], the P-RAM is therefore a "MIMD" ... "*

Stephen Cook express the same view:

> *[Coo81] page 118: "The P-RAM of Fortune and Wyllie ... different parallel processors can be executing different parts of their program at once, so it is "multiple instruction stream".*

The PRAM model is asynchronous

In [Sab88] pp. 162–163 the author compares his own model for parallel programming with the PRAM model. He starts by stating that the processors in the PRAM model operates asynchronously—which certainly is *not* the case for the original PRAM model. Further he writes (about the PRAM model) that it "... *is confusing and hard to program.*". (Probably, it is asynchronous shared memory multiprocessors the author considers as the "PRAM model".)

The author states that the (asynchronous) MIMD power is overwhelming, and that PRAM programmers deal with this power by writing SIMD style programs for it; he claims that "*all* parallel algorithms in the proceedings of the ACM Symposia on Theory of Computing (STOC) 1984, 85 and 86—are of data parallel form" (*i.e.* SIMD). This is *not* true. Algorithms in the proceedings of the STOC (and other similar) Symposia are typically described in a *high-level mathematical notation—where the choice between SIMD or MIMD programming style to a large extent is left open to the programmer.* Parts of an algorithm are most naturally coded in SIMD style, while other parts most naturally are expressed in MIMD style.

3.2.2 Notation: Parallel Pseudo Pascal—PPP

Wyllie outlined a high level pseudo code notation, called *parallel pidgin algol*, for expressing algorithms on a CREW PRAM. The notation introduced here, called Parallel Pseudo Pascal (PPP) is intended to be a modernisation of Wyllie's notation. PPP is an attempt to combine the pseudo language notation (called "super pascal") used for sequential algorithms by Aho, Hopcroft and Ullmann in [AHU82], with the ability to express parallelism and processor allocation as it is done in parallel pidgin algol.

Informal specification of PPP

The PPP notation is outlined in Table 3.1. Since PPP is *pseudo* code it should only be used as a *framework* for expressing algorithms: The PPP programmer should feel free to extend the notation and use own constructions. (See the last statement in the list below.)

A Statement may be any of the alternatives shown in the table, or a list of statements separated by semicolons and enclosed within **begin** ...**end**. Variables will normally not be declared, their definition will be implicit from the context or explicitly described.

It is crucial to separate local variables from global variables. This is done in PPP by underlining global variables. Local variables and procedure names are written in *italics*.

Statement 1 to 6 should be self-explaining, and statement 9 should be familiar to all which have experienced pseudo code. Here, only statement 7 and 8 need some further explanation.

Table 3.1: Informal outline of the Parallel Pseudo Pascal (PPP) notation.

1. (a) *LocalVariableName* := Expression
   (b) GlobalVariableName := Expression
2. **if** Condition **then** Statement1 { **else** Statement2 }
3. **while** Condition **do** Statement
4. **for** Variable := InitialValue **to** FinalValue **do** Statement
5. **procedure** *ProcedureName* (FormalParameterList) Statement
6. *ProcedureName* (ActualParameterList)
7. **assign** ProcessorSpecification[3]{, **to** ProcSetName }
8. **for each processor** ProcessorSpecification[4]{ **where** Condition } **do** Statement
9. any other well defined statement

Processor allocation
The **assign** statement is used to allocate processors. An example might be "**assign** a processor to each element in the array $\underline{\mathrm{A}}$, **to** Aprocs"

The **to** ProcSetName clause may be omitted in simple examples when only

one set of processors is used. The **assign** statement is realised by the processor allocation procedure (Section 3.1.2.2) which assigns local processor numbers to the processors. Thus processor no i may refer to the i'th element in the global array $\underline{\mathrm{A}}$ by $\underline{\mathrm{A}}[p]$ since $p = i$. (Remember that p is the local variable which always stores the processor number). The actual number of processors allocated by the statement may be run time dependent.

Processor activation
Statement type 8 is the way to specify that code should be executed in parallel. The processor specification may simply be the keyword **in** followed by the name of a processor set—in that case all the processors in that processor set will execute the statement in parallel.

The **where** clause is used to specify a condition which must be true for a processor in the processor set for the statement to be executed by that processor. A (rather low-level) PPP example (which might have been taken from a PPP program of an odd-even transposition sort algorithm) is shown in Figure 3.3.

```
for each processor in Aprocs where $p$ is odd and $p < n$ do begin
    $temp := \underline{\mathrm{A}}[p+1]$;
    if $temp < \underline{\mathrm{A}}[p]$ then begin
        $\underline{\mathrm{A}}[p+1] := \underline{\mathrm{A}}[p]$; $\underline{\mathrm{A}}[p] := temp$;
    end;
end;
```

Figure 3.3: Example program demonstrating the **where** clause. $\underline{\mathrm{A}}$ is a global array, *temp* and n are local variables, and p is (a local variable) holding the processor number.

[3] In this case, a ProcessorSpecification may be a text which implicitly describes a number of processors to be allocated, or simply "IntegerExpression processors".

[4] Here, ProcessorSpecification may be prose (describing text), or more formally "**in** ProcSetName".

3.2.3 The Importance of Synchronous Operation

Synchronous operation simplifies
Parallel programs are in general difficult to understand. However, requiring that the processors operate synchronously may often make it much easier to understand parallel algorithms. Consider the simple program in Figure 3.4. This example is taken from [Wyl79].

```
(1) for each processor in ProcSet do
(2)     if cond(p) then
(3)         x := f(y);
        else
(4)         y := g(x);
```

Figure 3.4: Example program with unpredictable behaviour.

The processor activation statement (line (1)) describes that the **if** statement shall be executed in parallel by all processors in the set called ProcSet. Assume that ProcSet contains two processors. Further assume that one processor evaluates the condition "cond(p)" to `true` while the other evaluates it to `false`. (This may happen because the condition may refer to variables local in each processor, or to the processor number). $\underline{\text{x}}$ and $\underline{\text{y}}$ are global variables.

Now consider the **else** clause. Depending on the relative times required to compute the functions f and g, either the old or new value of $\underline{\text{x}}$ will be used in the **else** clause. Thus it is difficult to argue about the precise behaviour of this program. As Wyllie points out [Wyl79], the code generated from the program in Figure 3.4 is a legitimate program for the P-RAM model—but this kind of programming is not recommended.

A program with more self-evident behaviour can easily be obtained from Figure 3.4, as shown in Figure 3.5. (It is here assumed that "cond(p)" is unaffected by lines (3) and (4).)

Line (5) describes an explicit *synchronisation* which is necessary to guarantee that the old values of $\underline{\text{x}}$ and $\underline{\text{y}}$ are used in the computation of f and g. Such synchronisations are in general valuable tools to restrict the space of possible behaviours and make it easier to understand parallel programs. This observation probably led Wyllie to define what is the most important property of (high-level) programming of the P-RAM;

```
(1) for each processor in ProcSet do begin
(2)     if cond(p) then
(3)         tx := f(y);
        else
(4)         ty := g(x);
(5)     wait for both processors to reach here;
(6)     if cond(p) then
(7)         x := tx;
        else
(8)         y := ty;
    end;
```

Figure 3.5: Example program transformed to predictable behaviour by using explicit resynchronisation (line (5)). (Line (5) may be omitted from a PPP program—see the text.)

CREW PRAM property 9 *(Synchronous statements)*
If each processor in a set P begins to execute a PPP statement S at the same time, all processors in P will complete their executions of S simultaneously. (Adapted from [Wyl79] p. 26) •

This surely is a severe restriction which puts a lot of burden on the programmer. However, the property is crucial in making the PPP programs understandable and easy to analyse. From now on, we will assume the existence of a PPP compiler that, whenever possible, automatically ensures that the PPP statements are executed as *synchronous statements.* Therefore we may omit statement (5) in Figure 3.5 since the PPP compiler on reading the **if** statement in line (2–4) will produce code so that all processors in ProcSet simultaneously will start executing the **if** statement in line (6–8).

Given the property *synchronous statements*, the time needed to execute a general (probably compound) PPP statement S is the maximum over all processors executing S plus the time needed to resynchronise the processors before they leave the statement.

The need for resynchronisation after each high-level statement may seem to give a lot of overhead associated with the high-level programming. As will be shown below, this is not the case. A very simple resynchronisation code may be added automatically by the compiler in most cases.

3.2.4 Compile Time Padding

Assuring the *synchronous statements* property may in most cases be done by so called *compile time padding.* When the PPP compiler knows the execution time of the **then** and **else** clauses—inserting an appropriate amount of waiting ("No op's") in the shortest of these will make the whole **if** statement to a synchronous statement.

Automatic compile time padding is in fact a necessity for making it possible to express algorithms in a synchronous *high level* language. The exact execution times of the various constructs should be hidden in a high level language. Neither is it desirable, nor should it be possible that the property of "synchronous statements" is preserved by the programmer.

Compile time padding should be used wherever it is possible by the compiler. It is simple, and it does not increase the execution time of the produced code. (It is only the uncritical execution paths which are padded till they have the same length as the critical (longest) execution path.) The alternative, which is to insert code for explicit resynchronisation, may be simpler for the compiler—but should be avoided since it introduces a significant increase in the running time, see Section 3.2.5.

3.2.4.1 "Helping the Compiler"

There are cases where compile time padding cannot be used, but where the use of general processor resynchronisation can be avoided by clever programming. Consider the following example[5] taken from [Wyl79]. The program in Figure 3.6 is a simple-minded program to set an array $\underline{\mathrm{A}}$ of non-negative integers to all zeros.

In this example it is not possible for the compiler to know the execution time of the while loop. It must therefore insert code for general resynchronisation just after the while loop. This may lead to a substantial increase in the execution time.

However, if the programmer knows the value of the largest element in the array $\underline{\mathrm{A}}$, $\underline{\mathrm{M}} = \max_i(\underline{A}[i])$, this fact may be used to help the compiler as shown in the program in Figure 3.7. Here the compiler knows that we will get $\underline{\mathrm{M}}$ iterations of the **for** loop for all the processors. Further it can easily make the **if** statement synchronous by inserting an **else** wait t; where t is

[5]A less artificial example is given on page 75.

```
begin
    assign a processor to each element in $\underline{A}$;
    for each processor do
        while $\underline{A}[p] > 0$ do
            $\underline{A}[p] := \underline{A}[p] - 1$;
end;
```

Figure 3.6: Example program which requires general resynchronisation to be inserted by a PPP compiler.

```
begin
    assign a processor to each element in $\underline{A}$;
    for each processor do
        for $i := 1$ to $\underline{M}$ do
            if $\underline{A}[p] > 0$ then
                $\underline{A}[p] := \underline{A}[p] - 1$;
end;
```

Figure 3.7: Example program where the need for general resynchronisation has been removed.

the time used to execute the **then** clause. All processors will simultaneously exit from the **for** loop.

3.2.4.2 Synchronous MIMD Programming—Example

Assume that we are given the task of making a synchronous MIMD program which computes the total area of a complex surface consisting of various objects. We will consider three cases that illustrate various aspects of synchronous MIMD programming.

Case-1:
Assume that the surface consists of only three kinds of objects, `Rectangle`, `Triangle` or `Trapezium`. The program is sketched in PPP in Figure 3.8.

Statement (2) demonstrates a multiway branch statement which utilise the multiple instruction stream property to give faster execution and better

[6] See for instance [HS86] or [KR88].

```
        ...
(1)     { Each processor holds one object }
(2)     if type of object is Rectangle then
            Compute area of Rectangle object
        else if type of object is Triangle then
            Compute area of Triangle object
        else
            Compute area of Trapezium object;
        { All processors should be here at the same time unit }
(3)     Compute total sum; { $O(\log n)$ standard parallel prefix[6]}
        ...
```

Figure 3.8: Synchronous MIMD program in PPP, case-1.

```
          +--- Rectangle ---: wait -----+
          |                             |
before -> {--- Triangle ---no wait -----} -> after
          |                             |
          +--- Trapezium ------: wait --+
```

Figure 3.9: Compile time padding.

processor utilisation than on a SIMD machine. If the time used to compute the area of each kind of object is roughly equal, MIMD execution will be roughly three time faster than SIMD execution for this case with three different objects.

As expressed by the comment just before statement (3), the processors should leave the multiway branch statement simultaneously. In this case this is easily achieved by compile time padding since the time used to compute the area of each kind of object may be determined by the compiler. If we assume that the most lengthy operation is to calculate the area of `Triangle` the effect of compile time padding can be illustrated as in Figure 3.9.

Case-2:
Now, assume that each object is a polygon with from 3 to 10 edges. As shown in Figure 3.10 the area of each object (polygon) is computed by a

```
        ...
        { Each processor holds one polygon }
(1)     Area := ComputeArea(this polygon);
        { All processors should be here at the same time unit }
(2)     Compute total sum; { $O(\log n)$ standard parallel prefix }
        ...
```

Figure 3.10: Synchronous MIMD program in PPP, case-2.

```
        ...
        { Each processor holds one polygon }
(1)     Area := ComputeArea(this object);
(2)     SYNC; { $O(\log n)$ time }
        { All processors should be here at the same time unit}
(3)     Compute total sum; { $O(\log n)$ standard parallel prefix }
        ...
```

Figure 3.11: Synchronous MIMD program in PPP, case-3.

function called *ComputeArea*. Knowing the maximum number of edges in a polygon, the compiler may calculate the maximum time used to execute the procedure, and produce code which will make all calls to the procedure use this maximum time.[7]

In many cases the compiler may calculate the maximum time needed by a procedure and insert code to make the time used on the procedure to a fixed, known quantity. This is important to provide high-level *synchronous* programming (CREW PRAM Property 9).

Case-3:
We now consider the case that each object is an arbitrary complex polygon (*i.e.* with an unknown number of edges), and that the size of the most com-

[7]Technically, this enforcing of the maximum time consumption may be done by inserting dummy operations and/or iterations, or by measuring the time used by reading the global clock.

plex polygon is not known by the processors.[8] In this situation it is impossible for the compiler to know the maximum execution time of *ComputeArea*. The best it can do is to insert an explicit synchronisation (statement (2)) just after the return from the procedure. `SYNC` is a synchronisation mechanism provided by the CREW PRAM simulator, and it is described in the next subsection. It is able to synchronise n processors in $O(\log n)$ time. In this case the use of `SYNC` does not influence on the time complexity of the computation, since the succeeding statement (3) also requires $O(\log n)$ time.

3.2.5 General Explicit Resynchronisation

There are cases where general resynchronisation cannot be avoided. In general, the number of processors leaving a statement must match the number of processors which entered it. The number of entering processors is known at run time, so the task reduces to count the number of processors which have finished the statement and are ready to leave it.

One of the main purposes of our CREW PRAM model simulator is to be a vehicle for polylogarithmic algorithms. Many such algorithms use at least $O(n)$ processors where n is the problem size. The use of a simple synchronisation scheme with a linear ($O(n)$) time requirement would make it impossible to implement parallel algorithms using explicit processor resynchronisation with sublinear time requirement.

However, as Wyllie claims, synchronisation of n processors may be done in $O(\log n)$ time. He does not describe the implementation details, but they are rather straightforward. One implementation is outlined in Section 3.2.5.1.

Since processor synchronisation may be necessary in CREW PRAM programs, the time requirement modelling should use a realistic measure for the time used by resynchronisations. One will soon realise that the time needed by (most) resynchronisation algorithms depends on when the various processors starts to execute the algorithm. Thus, the time requirement cannot be computed as an exact function of solely the number of synchronising processors. One way to exactly model the time usage is to implement the synchronisation.

[8]Knowing the size of the most complex polygon would make it possible to use the techniques described for *case-2*.

3.2.5.1 Synchronisation Algorithm

We will now outline the implementation of processor resynchronisation which is used in the CREW PRAM simulator. It should be noted that this is just one of many possible solutions. It may be omitted by readers who are not interested in what happens below the high-level language (PPP).

Assume that n processors finish a statement at unpredictable, different times. When all have finished, they should, as fast as possible, proceed simultaneously (synchronously) to the next statement. Before this happens, they must wait and synchronise.

As is often the case for $O(\log n)$ algorithms, the necessary computation may be done by organising the processors in a binary tree structure. Assume that the processors are numbered $1, 2, \ldots n$. The waiting time may be used to let each processor count the number of processors in the subtree (below and including itself) which have finished the statement. This is done in parallel by all (finished) processors. The number of finished processors will "bubble" up to the root of the whole tree, which holds the total number of finished processors.

When all have started to execute the resynchronisation algorithm, the time used before this is known by the root processor is given by the time used to propagate the information in parallel from the leaves to the root of the tree. This is surely $O(\log n)$. A similar $O(\log n)$ time algorithm is the *synchronisation barrier* reported in [Lub89].

Figure 3.12 illustrates the performance of the resynchronisation algorithm for processor sets of various sizes (number of processors). For each size ten cases have been measured. In each case, every processor uses a random number of time units, in the range $[1, 100]$, before it calls the synchronisation procedure. The synchronisation time is measured as the time from the last processor calls the synchronisation procedure to all the processors have finished the synchronisation.

3.3 Parallelism Should Make Programming Easier!

This section describes some of the thoughts presented at the Workshop On The Understanding of Parallel Computation under the title *Parallelism Should Make Programming Easier!* [Nat90c].

The motivation for the section is two-fold. First of all, it gives a brief explanation of my opinion that parallelism may be used to make programming easier, and it shortly mentions some related work. Secondly, two examples of "easy parallel programs" are provided. These examples illustrate the use of the PPP notation introduced in the previous section.[9]

3.3.1 Introduction

Parallelism should not add to the "software crisis"
The so-called "software crisis"[10] is still a major problem for computer industry and science. Contemporary parallel computers are in general regarded to be even more difficult to program than sequential computers. Nevertheless, the need for high performance has made it worthwhile to develop complex program systems for special applications on parallel computers.

The use of parallelism should not add to the "software crisis". For general purpose parallel computing systems to become widespread, I think it is necessary do develop programming methods that strive for making (parallel) programming easier.

How can parallelism make programming easier?
There are at least two main reasons that the use of parallelism in future computers may make programming easier.

The main motivation for using parallel processing is to provide more computing power to the user. This power may in general make it possible for the programmer to use simpler and "dumber" algorithms—to some extent. It is well know that coding to achieve maximum efficiency is difficult and (programmer)-time consuming. Consequently, *parallelism may increase the number of cases where we can afford to use simple "brute force" techniques.* Also, increased computing power (or a large number of processors) may

[9]The use of the PPP notation is the reason for placing this relatively "high-level" section between two technical sections (3.2 and 3.4).

[10]Booch in [Boo87] page 7": "The essence of the software crisis is simply that it is much more difficult to build software systems than our intuition tells us it should be".

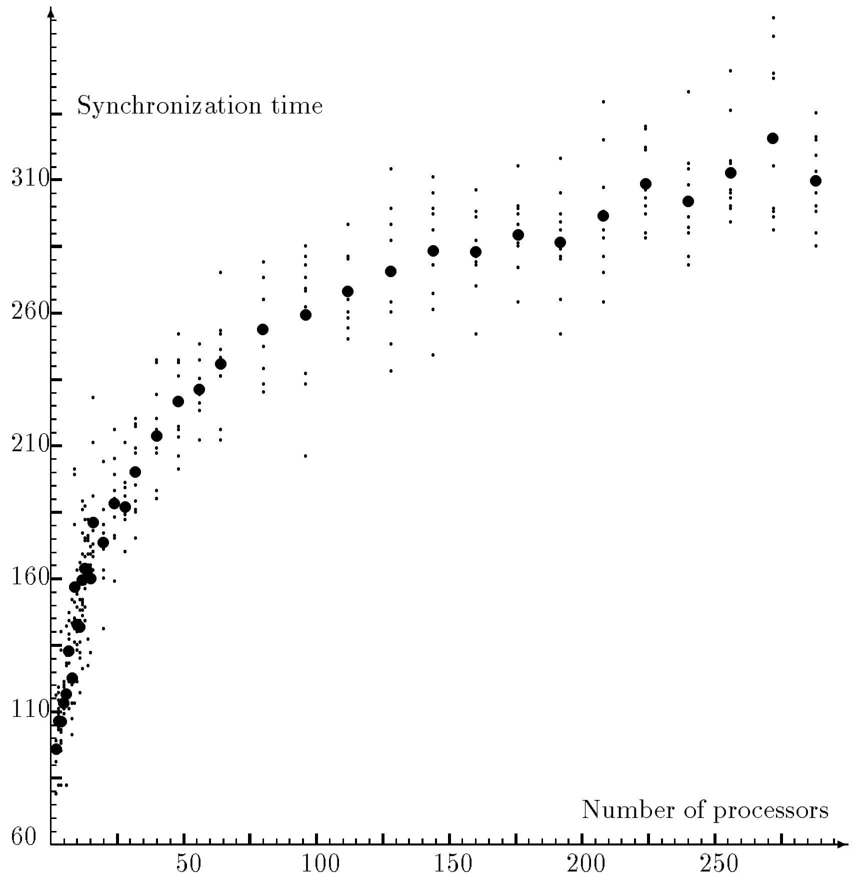


Figure 3.12: Performance of the resynchronisation algorithm provided by the CREW PRAM simulator. The symbol • is used to mark the average time used of ten random cases. A · is used to mark each random case. (See the text.)

make it more common to use "redundancy" in problem solving strategies—which again may imply conceptually simpler solutions. (See the example in Section 3.3.2.1 below).

Many of our programs try to reflect various parts of, or processes in, the real world. There is no doubt that the real world is highly parallel. Therefore, *the possibility of using parallelism will in general make it easier to reflect real world phenomena and natural solution strategies.* (See the example in Section 3.3.2.2 below).

Related work

Most computer scientists would of course prefer easy programming. Consequently, there have been made several proposals of programming languages that aim at providing this. In this paragraph, I will mention some of these to provide a few pointers to further reading. I will not judge the relative merits of the various approaches.

The August 1986 issue of IEEE COMPUTER was titled *"Domesticating Parallelism"* and contains articles describing several of these new languages; Linda [ACG86], Concurrent Prolog [Sha86], ParAlfl (a parallel functional language) [Hud86] and others.

Linda consists of a small set of primitives or operations that is used by the programmer to construct explicitly parallel programs. It supports a style of programming that to a large extent makes it unnecessary to think about the coupling between the processors. (See also [CG88].)

Concurrent Prolog is just one of several logic programming languages designed for parallel programming and execution. Numerous parallel variants of LISP have also been developed, one of the first was *Multilisp* [Gel86, Hal85].

Strand is a general purpose language for concurrent programming [FT90] and is currently available on various parallel and uni-processor computers.

UNITY by Chandy and Misra [CM88] is a new programming paradigm that clearly separates the program design, which should be machine independent, from the mapping to various architectures, serial or parallel. (See also [BCK90]).

SISAL is a so-called single assignment language derived from the dataflow language VAL. Unlike dataflow languages, SISAL may be executed on sequential machines, shared memory multiprocessors, and vector processors as well as dataflow machines [OC88]. See also [CF90] and [Lan88].

For traditional, sequential programming, there is far from any consensus about which of the various programming paradigms being most successful in providing easy programming. We must therefore expect similar discussions for many years about the corresponding parallel programming paradigms.

A very important approach to avoid that the use of parallelism will *add* to the software crisis is simply to let the programmers continue to write sequential programs. The parallelising may be left to the compiler and the run-time system. A lot of work has been done in this area, see for instance [Pol88, Gup90, KM90]. The method has the great advantage that old programs can be used on new parallel machines without rewriting.

3.3.2 Synchronous MIMD Programming Is Easy

The use of synchronous MIMD programming to implement the algorithms studied and described in this thesis was motivated by a wish to program in a relatively high level notation which was consistent with the "programming philosophy" used by James Wyllie [Wyl79]. It came as a surprise that high-level synchronous MIMD programming on a CREW PRAM model seems to be remarkably easy.

Easy programming
The CREW PRAM property *Synchronous statements* described at page 73 implies that the benefits of synchronous operation are obtained at the "source code level", and not only at the instruction set level. It gives the advantages of SIMD programming, *i.e.* programs that are easier to reason about [Ste90]. However, we have kept the flexibility of MIMD programming. This was illustrated by the example in Section 3.2.4.2, where Figure 3.8 shows a MIMD program that may compute the area of three different kinds of objects simultaneously. Such natural and efficient handling of conditionals is not possible within the SIMD paradigm.[11]

At first, making the statements synchronous may be perceived as a heavy burden to the programmer. However, my experience is that it is a relatively easy task after some training, and that it may be helped a lot with simple tools.

Note that the synchronous operation property may be violated by the programmer if wanted. Asynchronous behaviour can be modelled on a CREW PRAM—with the only restriction that the time-difference between

[11]On a SIMD system, nested conditionals can in principle reduce processor utilisation exponentially in the number of nesting levels [Ste90].

any two events must be a whole number of CREW PRAM time units. This "discretisation" should not be a problem in practice.

Easy analysis

Synchronous programs exhibit in general a much more deterministic behaviour than asynchronous programs. This implies easier analysis.

The process of making high level statements synchronous does also simplify analysis. This is demonstrated by *case-1* of the example in Section 3.2.4.2. In that example, compile time padding was used to assure that a computational task would use an equal amount of time for three different cases. Here, redundant operations (*i.e.* wait-operations) simplify.

Making high level statements synchronous may also make it necessary to code a procedure such that its time consumption is made *independent* of its input parameters. A natural approach is then to code the procedure for solving the most general case, and to use this code on all cases. This removes the possibility of saving some processor time units on simple cases—but more important, it simplifies the programming of the procedure.

A consequence of synchronous statements is that the time consumption of an algorithm will typically be less dependent on the actual nature of the problem instance, such as presortedness [Man85] for sorting. This makes it easier to use the algorithm as a substep of a larger synchronous program.

Easy debugging

> "The main problems associated with debugging concurrent programs are increased complexity, the "probe effect", nonrepeatability, and the lack of a synchronised clock."
>
> McDowell and Helmbold in
> *Debugging Concurrent Programs* [MH89].

The debugging of the synchronous MIMD programs developed as part of the work reported in this thesis has been quite similar to debugging of uniprocessor programs. The synchronous operation and the execution of the programs on a simulator have eliminated most of the problems with concurrent debugging. The CREW PRAM model implicitly provides a synchronised clock known by all the processors. Synchronous operation will in general imply an increased degree of repeatability, and full repeatability is possible on the simulator system. The simulator does also provide monitoring of the programs without any disturbance of the program being evaluated—*i.e.* no "probe effect".

Not all of these benefits would be possible to obtain on a "CREW PRAM machine". However, a proper programming environment for such a machine should contain a simulator for easy debugging in "early stages" of the software testing.

Having experienced the benefits of synchronous MIMD programming after only a few months of "training" and with the use of very modest prototype tools, I am convinced that this paradigm for parallel programming is worth further studies in future research projects.

3.3.2.1 Example-1, Odd-even Transposition Sort

The purpose of this example is to illustrate how parallelism through increased computing power and redundancy may give a faster *and simpler* algorithm. We know that sorting on a sequential computer can not be done faster than $O(n \log n)$ [Akl85, Knu73]. This is achieved by several algorithms, for instance *Heapsort* invented by [Wil64]. (The famous *Quicksort* algorithm invented by C. A. R. Hoare [Hoa62] is $O(n \log n)$ time on *average*, but only $O(n^2)$ in the worst case [AHU82].)

However, if we have n processors available for sorting n items, it is very easy to sort faster than $O(n \log n)$ time if we allow a higher cost than $O(n \log n)$. $O(n)$ time sorting is easily obtained by several $O(n^2)$ cost parallel sorting algorithms.

Perhaps the simplest parallel sorting algorithm is the *odd-even transposition sort* algorithm. The algorithm is now being attributed [Qui87, Akl89] to Howard Demuth and his Ph.D. thesis from 1956. See [Dem85]. The algorithm is so well known that I will only outline it briefly. Readers unfamiliar with the algorithm are referred to one of [Qui87, Akl85, Akl89, BDHM84] for more detailed descriptions.

The odd-even transposition sort algorithm is based on allocating one processor to each element of the array which is to be sorted. See the example in Figure 3.13 which shows the sorting of 4 items. If n, the number of items to be sorted is an even number, we need $n/2$ iterations. Each iteration consists of an odd phase followed by an even phase. The processors are numbered by starting at 1. In the odd phase, each processor with odd number compares its element in the array with the element allocated by the processor on the

```
Iteration 1:     4  2  3  1
                 | /   | /
   odd phase     ?     ?

                 2  4  1  3
                    | /
   even phase       ?
Iteration 2:     2  1  4  3
                 | /   | /
   odd phase     ?     ?

                 1  2  3  4
                 | /   | /
   even phase    ?     ?
```

Figure 3.13: Sorting 4 numbers by odd even transposition sort. The operation of a processor comparing two numbers are marked with ?.

right side. In the example, processor 1 compares 4 with 2, and processor 3 compares 3 with 1. If a processor finds that the two elements are out of order, they are swapped. In even phases, the same is done by the even numbered processors.

A PPP program for the algorithm is given in Figure 3.14. Note that $n-1$ processors are needed.[12]

I would guess that most readers familiar with this algorithm or similar parallel algorithms will agree that it is simpler than heapsort and quicksort. In my opinion, it is also simpler than straight insertion sort—which is commonly regarded to be one of the simplest uni-processor sorting algorithms (see the next example).

3.3.2.2 Example-2, Parallel Insertion Sort

This example is provided to show how the use of parallelism may make it easier to reflect a natural solution strategy.

One of the simplest uniprocessor algorithms is straight insertion sort. This is the method used by most card players [Wir76]. A good description is found in Bentley's book *"Programming pearls"* [Ben86].

The method is illustrated in Figure 3.15. One starts with a sorted sequence of length one. The length of this sequence is increased with one

[12]The reader may have observed that only $n/2$ processors are active at any time during the execution of the algorithm. A slightly different implementation that takes this into account is described in Section 3.4.

```
assign $n - 1$ processors to ProcArray;
for each processor in ProcArray begin
    for $Iteration := 1$ to $n/2$ do begin
        if $p$ is odd then
            if $\underline{\mathrm{A}}[p] > \underline{A}[p+1]$ then Swap($\underline{\mathrm{A}}[p]$, $\underline{\mathrm{A}}[p+1]$);
        if $p$ is even then
            if $\underline{\mathrm{A}}[p] > \underline{A}[p+1]$ then Swap($\underline{\mathrm{A}}[p]$, $\underline{\mathrm{A}}[p+1]$);
    end;
end;
```

Figure 3.14: Odd even transposition sort expressed in the PPP notation.

```
start:               3 *1  4  2     <sorted> * <unsorted>
after iteration 1:   1  3 *4  2
after iteration 2:   1  3  4 *2
after iteration 3:   1  2  3  4
```

Figure 3.15: Straight insertion sort of 4 numbers.

```
(T)      for $Next$ := 2 to $n$ do begin
             { Invariant: $\underline{A}[1..Next-1]$ is sorted }
(T)          $j := Next$;
(F,M,I)      while $j > 1$ and $\underline{A}[j-1] > \underline{A}[j]$ do begin
                 Swap($\underline{A}[j]$, $\underline{A}[j-1]$);
                 $j := j - 1$;
             end;
         end;
```

Figure 3.16: Straight insertion sort with one processor [Ben86].

number in each iteration. Each iteration can be perceived as consisting of four tasks:

(T) *Take* next. This corresponds to picking up the card to the right of the $*$ in Figure 3.15.

(F) *Find* the position where this card should be inserted in the sorted sequence (the numbers to the left of $*$).

(M) *Make* space for the new card in the sorted sequence.

(I) *Insert* the new card at the new free position.

The algorithm is shown in Figure 3.16. The code corresponds to the simplest version of the algorithm presented at top of page 108 in Bentley's book [Ben86]. To the left of each statement the letters 'T', 'F', 'M', and 'I' have been used to show how the various parts of this code model the four main tasks just described. The three tasks of finding the right position, making space and inserting the next card are all taken care of by the simple while loop.

In my opinion, the straight insertion sort algorithm, as performed by card players, is easier to model and represent by using parallelism. See Figure 3.17. Processor p is allocated to $\underline{A}[p]$. The position of the next card to be inserted in the sorted subsequence is stored in the variable *Next* which is local to each processor. The sorted subsequence is represented by the processors to the left of *Next*. Statement [a] assures that the following code is performed only by the processors allocated to numbers in the *sorted*

```
         assign $n-1$ processors to ProcArray;
         for each processor in ProcArray
(T)          for $Next := 2$ to $n$ do
[a]              if $p < Next$ then begin
                     $NextVal := \underline{\text{A}}[Next]$;
(F)[b]               if $NextVal < \underline{\text{A}}[p]$ then begin
(M)[c]                   $\underline{\text{A}}[p+1] := \underline{\text{A}}[p]$;
(F)[d]                   if $NextVal \geq \underline{\text{A}}[p-1]$ then
(I)[e]                       $\underline{\text{A}}[p] := NextVal$;
                     end;
                 end;
```

Figure 3.17: Parallel CREW PRAM implementation of straight insertion sort expressed in PPP. (See the text.)

sequence. Each of these processors start by reading the value of the next card into the local variable *NextVal*.

The two statements marked (F) model how the right position for inserting the next card is found. Statement [*b*] selects all cards in the sorted sequence with higher value than the next card. To make space for the next card, these are moved one position to the right, as described by statement [*c*]. Statement [*d*] specifies that if the value of the next card is larger (or equal) to the card on the left side ($\underline{\text{A}}[p-1]$), then we have found the right position for inserting the card (Statement [*e*]).

I will not claim that this program is easier to understand, but it seems to me that its behaviour does more clearly reflect the sorting strategy performed by card players. The "pattern matching" used by card players to find the right position for insertion, and especially the making of space for the next card are typical parallel operations.[13] (I hope to find more striking examples in the future!).

[13]See also the VLSI parallel shift sort algorithm by Arisland et al. [AAN84].

3.4 CREW PRAM Programming on the Simulator

Introduction

This section outlines an approach for top down development of synchronous MIMD programs in the CREW PRAM simulator environment. It is placed in this chapter since it describes CREW PRAM programming. However, it does refer to technical details of the CREW PRAM simulator which is described in Appendix A. *For a detailed study, Appendix A should be read first. Alternatively, use the index provided at the end of the thesis.*

It should be noted that this approach is not an ideal approach for synchronous CREW PRAM programming, because it reflects limitations of the used CREW PRAM simulator prototype.

The recommended development approach

The approach for top down development is a natural extension of the top down stepwise refinement strategy often used when developing sequential algorithms. It is explained how early, incomplete versions of the program may be made and kept synchronous in a convenient manner. Further, it is described why the exact modelling of the time consumption should be postponed to the last stage in the development process.

The approach is illustrated by showing four possible steps (*i.e.* representations) in the development of an odd-even transposition sort CREW PRAM algorithm. More complicated algorithms will probably require a further splitting of one or several of these steps.

3.4.1 Step 1: Sketching the Algorithm in Pseudo-Code

The algorithm used as an example for outlining the development approach is a slightly different version of the odd-even transposition sort algorithm presented in Section 3.3.2.1. In that version only one half of the processors are participating in each stage of the algorithm, the other half is waiting. That implementation, using $n-1$ processors, was chosen because it is very similar to the odd-even transposition sort algorithms presented for a linear array of n processors, such as [Akl85] pp. 41–44, [Qui87] pp. 88–89, or [Akl89] pp. 89–92.

However, doing odd-even transposition sort on a CREW PRAM, there is no need to use more than $n/2$ processors. The $n/2$ processors may act as "odd" and "even" processors in an alternating style. Processor number

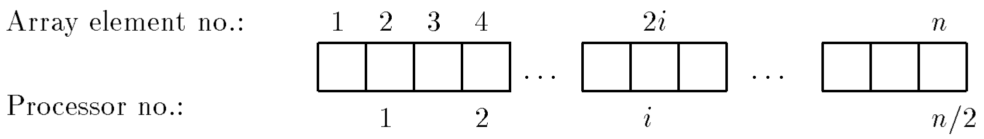


Figure 3.18: Processor allocation using $n/2$ processors for odd-even transposition sort of n elements.

i may be assigned to the element in position $2i$ of the array which is to be sorted, see Figure 3.18. In even numbered stages, processor i acts as an even numbered processor assigned to element $2i$, and compares this element with the element in position $2i+1$. In odd numbered stages, processor i acts as an odd numbered processor assigned to the element $2i-1$, and compares that element with the element in position $2i$. High level pseudo code for the algorithm, expressed in the PPP notation, is shown in Figure 3.19.

3.4.2 Step 2: Processor Allocation and Main Structure

A crucial part of every parallel algorithm is the *processor allocation* and *activation*. The exact number of processors that will be used should be expressed as a function of the problem size. Further, the *main structure* of the algorithm including the activation of the processors should be clear at this stage.

A possible first version of a program for our odd-even transposition sort algorithm is shown in Figure 3.20. This is a complete program that may be compiled and executed on the simulator. It is written in a language called PIL (see Section A.2.3.1) which is based on SIMULA [BDMN79, Poo87].

The program demonstrates various essential features of PIL. The `#include` statement includes a file called `sorting` which contains the procedure `GenerateSortingInstance`. The assignment of processors and processor activation are done by statements which are quite similar to the PPP notation. The `READ n FROM ...` statement shows the syntax for reading from the global memory. (The corresponding `WRITE` statement is shown in Figure 3.22.) `T_ThisIs` outputs the identification of the calling processor and is used for debugging. `T_TO` is a procedure for program tracing. The procedure call `Use(1)` specifies that each processor should consume one time

```
CREW PRAM **procedure** *OddEvenSort*
begin
    assign $n/2$ processors, to ProcArray;
    for each processor in ProcArray begin
        Initialise local variables;
        for *Iteration* := 1 to $n$ do begin
            Compute address of array element associated with each
                processor during this iteration;
            Read that array element, and also its right neighbour
                element if it exists;
            Compare the two elements just read and swap
                them if needed;
        end;
    end;
end;
```

Figure 3.19: Pseudo code (PPP) for odd-even transposition sort using $n/2$ processors.

```
% oesNew1.pil
% Odd Even Transposition sort with n/2 processors.

#include<sorting>

PROCESSOR_SET ProcArray;
INTEGER n; ! problem size ;
ADDRESS addr;

BEGIN_PIL_ALG
  n := 10;
  addr := GenerateSortingInstance(n, RANDOM);
  ASSIGN n//2 PROCESSORS TO ProcArray;

  FOR_EACH PROCESSOR IN ProcArray
  BEGIN ! parallel block ;
    INTEGER Iteration;

    ! Initialize local variables ;
    ! n is written on the UserStack by GenerateSortingInstance ;
    READ n FROM UserStackPtr-1;

    FOR Iteration := 1 TO n DO
    BEGIN
      T_ThisIs;
      IF IsOdd(Iteration) THEN T_TO(" Odd iteration")
                          ELSE T_TO(" Even iteration");
      Use(1);
    END_FOR;

  END of parallel block;
  END_FOR_EACH PROCESSOR;

END_PIL_ALG
```

Figure 3.20: Main structure of PIL program for our odd-even transposition sort algorithm. It contains processor allocation and activation.

unit before it finishes the last iteration of the `FOR Iteration` ...loop. For the purpose of this program, this is sufficient time modelling. For further details, consult Section A.2.

3.4.3 Step 3: Complete Implementation With Simplified Time Modelling

The next main step in the development process should be to make a complete implementation of the algorithm. Normally, this should be done in several substeps. The resulting PIL program might be as shown in Figure 3.21 and 3.22. This version of the program implements the alternation of the processors between odd and even by adjusting the value of `ThisAddr` in each iteration. The two values to be compared by every processor in each iteration are read by the procedure `ReadTwoValues` and later processed by `CompareAndSwapIfNeeded`. `ArrayPrint` is a built in procedure which prints a specified area of the global memory—in this case the array of sorted numbers.

Note that *it is generally recommended to do only a simplified time modelling at this stage.* The exact modelling of the time used by the parallel algorithm should be postponed to the last stage of the development. There is no need to go into *full detail* with respect to the time consumed before a complete and correct version of the algorithm have been made. However, whenever it is possible, all early versions of the program should be made synchronous. The absence of proper compile time padding makes it necessary to use manual methods for achieving synchronous programs. This is discussed below.

Making programs synchronous using `SYNC`

The currently used PIL compiler `pilc` is unfortunately not able to perform compile time padding to make the statements synchronous. Therefore, this dull work must be done by the PIL programmer. However, the burden may to a large extent be alleviated by using a stepwise refinement strategy on the time modelling.

Consider the `IF` statement in the procedure `CompareAndSwapIfNeeded` in the PIL program in Figure 3.22. The `ELSE` clause could have been omitted by the programmer if the compiler was able to do compile time padding. The compiler would then automatically insert the `ELSE` clause to guarantee that all processors executing the `IF` statement would leave that statement

```
% oesNew2.pil
% Odd Even Transposition sort with n/2 processors,
% complete algorithm with simplified time modelling.

#include<sorting>

PROCESSOR_SET ProcArray;
INTEGER n;
ADDRESS addr;

BEGIN_PIL_ALG
  n := 10;
  addr := GenerateSortingInstance(n, RANDOM);

  ASSIGN n//2 PROCESSORS TO ProcArray;
  FOR_EACH PROCESSOR IN ProcArray
  BEGIN ! parallel block;
    INTEGER n, Iteration;
    INTEGER ThisVal, RightVal;
    ADDRESS ThisAddr;

#include<oesNew2.proc>

    ! n is written on the UserStack by GenerateSortingInstance ;
    READ n FROM UserStackPtr-1;
    ThisAddr := (UserStackPtr - 1 - n) + ((p*2)-1);

    FOR Iteration := 1 TO n DO
    BEGIN
      IF IsOdd(Iteration) THEN ThisAddr := ThisAddr - 1
                          ELSE ThisAddr := ThisAddr + 1;
      ReadTwoValues;
      CompareAndSwapIfNeeded;
    END_FOR;

  END of parallel block;
  END_FOR_EACH PROCESSOR;
  ArrayPrint(addr, n);
END_PIL_ALG
```

Figure 3.21: PIL main program for our odd-even transposition sort algorithm. The program contains a simplified time modelling which is sufficient to keep it synchronous. The included file `oesNew2.proc` is shown in Figure 3.22.

```
% oesNew2.proc
    PROCEDURE ReadTwoValues;
    BEGIN
      READ ThisVal  FROM ThisAddr;
      IF ThisAddr + 1 >= UserStackPtr - 1 THEN
      BEGIN
        Use(t_READ); RightVal := INFTY;
      END
      ELSE
        READ RightVal FROM ThisAddr+1;
    END_PROCEDURE ReadTwoValues;

    PROCEDURE CompareAndSwapIfNeeded;
    BEGIN
      IF RightVal < ThisVal THEN
      BEGIN
        WRITE RightVal TO ThisAddr;
        WRITE ThisVal TO ThisAddr+1;
      END
      ELSE ! Numbers compared are in right order, need not write;
        Wait(t_WRITE + t_WRITE); ! to preserve synchrony ;
    END_PROCEDURE CompareAndSwapIfNeeded;
```

Figure 3.22: Procedures used in Figure 3.21.

simultaneously. The parameter value in the `Wait` call[14] must be exactly equal to the time used in the `THEN` clause of the `IF` statement. It is necessary to recalculate this parameter each time a modification that changes the time consumption in some part of the (possibly nested) `THEN` clause is made.[15] Such modifications occur frequently during development of an algorithm, and the recalculation of this parameter is typical compiler work.

The only motivation for the `Wait` call is to assure that all processors leave the `IF` statement simultaneously. This property may be achieved in an alternative way: Insert a call to the general resynchronisation procedure `SYNC` just after the `IF` statement. This makes it possible to omit the `ELSE` part and the code executed in the `THEN` clause may be changed without destroying the synchronous property of the `IF` statement (considered as *one* statement including the appended call to `SYNC`).

The use of `SYNC`, will result in an increase in the time used by the algorithm. This is no problem since the *exact* time modelling have been postponed to a later stage.

[14]This parameter is written as `t_WRITE + t_WRITE` to reflect that it represents the time used by two succeeding `WRITE` statements. For the use of "symbolic constants" such as `t_WRITE` see also Section 3.4.4.

[15]In this simple example, the time used in the `THEN` clause is easy to calculate—however most real examples will imply more elaborate calculations.

Strive for a correct ordering of the global events

All time consumption may *not* be omitted from early versions. The crucial issue is to distinguish between local and global events. *Local events* are those occurring inside a processor. They cannot affect other processors, and they cannot be affected *by* other processors. *Global events*, such as access to the global memory, may affect (writes) or be affected by (reads) other processors.

> *Early versions of the program should use a simplified time modelling—but should try to obtain the same ordering of the global events that is expected in a complete implementation with exact time modelling.*[16]

For the correctness of an algorithm, it is irrelevant whether a set of completely local computations take one time unit or x time units, as long as the order of *all* global events implied by the algorithm is kept unaltered.

The `READ` and the `WRITE` statement both take one time unit (see page 180 in Section A.2.3.6). This time consumption is automatically modelled by the simulator. In addition, *the simplified time modelling* in Figure 3.22 consists of one call to `Wait` as discussed above, and a call `Use(t_READ);` in the procedure `ReadTwoValues`. The latter call ensures that all processors will leave that procedure synchronously.

3.4.4 Step 4: Complete Implementation With Exact Time Modelling

When a complete implementation of the algorithm has been tested and found correct—it is time to extend the program with more exact modelling of the time consumption. This is a relatively trivial task which may be split into two phases.

Specifying time consumption with `Use`

A complete program with exact time modelling for our example algorithm is shown in Figure 3.23. The main difference between this program and the previous version (Figure 3.21) is that a lot of calls to the `Use` procedure have been introduced. Note that this explicit specification of the time used implies the advantage that the programmer is free to assume any particular instruction set or other way of representing the time used by the program.

[16]This would in general be very difficult for an asynchronous algorithm. For synchronous programs it will generally be much easier.

```
% oesNew3.pil
% Odd Even Transposition sort with exact time modelling.

#include<sorting>

PROCESSOR_SET ProcArray;
INTEGER n;
ADDRESS addr;

BEGIN_PIL_ALG
  n := 10;
  addr := GenerateSortingInstance(n, RANDOM);
  ClockReset; !****** Start of exact time modelling ;

  ASSIGN n//2 PROCESSORS TO ProcArray;
  FOR_EACH PROCESSOR IN ProcArray
  BEGIN ! of parallel block;
    INTEGER n, Iteration;
    INTEGER ThisVal, RightVal;
    ADDRESS ThisAddr;

#include<oesNew3.proc>

    ! SetUp: ;
    Use(t_LOAD + t_SUB);
    READ n FROM UserStackPtr-1;
    Use((t_LOAD + t_SUB + t_SUB) + t_ADD +
        (t_LOAD + t_SHIFT + t_SUB) + t_STORE);
    ThisAddr := (UserStackPtr - 1 - n) + ((p*2)-1);

    Use(t_LOAD + t_SUB + t_JZERO);
    FOR Iteration := 1 TO n DO
    BEGIN
      Use(t_SHIFT + t_JZERO);
      IF IsOdd(Iteration) THEN
      BEGIN
        Use(t_SUB);
        ThisAddr := ThisAddr - 1;
      END
      ELSE
      BEGIN ! even Iteration;
        Use(t_ADD);
        ThisAddr := ThisAddr + 1;
      END;

      ReadTwoValues;
      CompareAndSwapIfNeeded;

      Use(t_LOAD + t_ADD + t_STORE + t_SUB + t_JZERO);
    END_FOR;
  END of parallel block;
  END_FOR_EACH PROCESSOR;

  ArrayPrint(addr, n);
END_PIL_ALG
```

Figure 3.23: Complete PIL program for odd-even transposition sort with exact time modelling. The included file `oesNew3.proc` is shown in Figure 3.24.

```
% oesNew3.proc

    PROCEDURE ReadTwoValues;
    BEGIN
      READ ThisVal  FROM ThisAddr;
      Use(t_ADD + t_IF);
      IF ThisAddr + 1 >= UserStackPtr - 1 THEN
      BEGIN
        Use(t_READ);
        RightVal := INFTY;
      END
      ELSE
        READ RightVal FROM ThisAddr+1;
    END_PROCEDURE ReadTwoValues;

    PROCEDURE CompareAndSwapIfNeeded;
    BEGIN
      Use(t_LOAD + t_SUB + t_JNEG);
      IF RightVal < ThisVal THEN
      BEGIN
        WRITE RightVal TO ThisAddr;
        Use(t_ADD);
        WRITE ThisVal TO ThisAddr+1;
      END
      ELSE ! Numbers compared are in right order, need not write;
        Wait(t_WRITE + t_ADD + t_WRITE); ! to preserve synchrony ;
    END_PROCEDURE CompareAndSwapIfNeeded;
```

Figure 3.24: Procedures used in Figure 3.23.

The following "working rules" may be useful when introducing more exact time modelling in a PIL program:

1. *Use "symbolic time constants"*
   When specifying time consumption as a parameter to `Use` or `Wait` the programmer should try to make also this part of the code as readable as possible. Using predefined constants such as `t_ADD`, `t_SHIFT` etc. is a simple way to document how the programmer imagines that the various parts of the high level PIL program would have been represented in CREW PRAM machine code.[17] Also, using such sums of symbolic constants instead of so-called magic numbers makes it easier to do the necessary modifications to the time modelling when changes to the PIL program are made.
   In larger programs, it may be practical to define constants that reflect the time used by various logical subparts.

2. *"Cluster local time consumption"*
   Bearing in mind the difference between local and global events discussed at page 97, the time used on several subsequent local computations may in general be collected into one call to `Use`. Consider for instance the `Use` call at the end of the `FOR` loop in Figure 3.23, which models the time used to control that loop.

3. *Work systematically from time zero*
   It is natural to introduce the exact time modelling by "following the code structure" from (execution) time zero in a top down, depth first approach.

4. *Do small changes and test systematically*
   Since the time used in various parts of a parallel system may affect the order of the global events, it may indirectly affect the correctness of the algorithm. Experience have shown that minimal changes of the timing may cause disastrous changes to the program. Therefore, the introduction of the exact time modelling should be split into several substeps—with systematically testing in between.

[17]These constants are defined in the simulator source file `times.sim` stored in the `src` directory (see Appendix A.2). They are used in central parts of the simulator and should therefore not be changed.

Replacing `SYNC` by `CHECK`

If exact time modelling has been introduced properly together with the necessary `Wait` calls for achieving synchronous operation, it should be possible to remove all *explicit* resynchronisation which have been inserted as discussed at page 96. Simply removing all the calls to `SYNC` may be too optimistic. It is likely that you will do future modifications to your program that may change the time used. Therefore, "far-sighted" programmers will replace `SYNC` by `CHECK`. `CHECK` checks whether the processors are synchronous at the given point, and may detect cases where "synchrony" have been lost due to erroneous program modifications. Replacing calls to `SYNC` with calls to `CHECK` should be done one by one, with testing after each replacement.

Closing remarks

The reader should have noticed that exact time modelling requires the specification of a lot of boring details that are strictly related to the details of the program implementing the algorithm. Considering the amount of changes normally done during the development of a program, it should be clear that a lot of unnecessary work may be avoided by postponing the exact time modelling to the very end of the program development. Please note that most of these details would have been taken care of by a proper PIL compiler.

Chapter 4

Investigation of the Practical Value of Cole's Parallel Merge Sort Algorithm

"A fundamental choice is whether to base the message routing strategy of such an emulation on sorting or to use a method that does not require sorting; and, if sorting is to be used, whether there exists an $O(\log n)$-time sorting method for which the constant hidden by the big-O is not excessive."

Richard M. Karp in *A Position Paper on Parallel Computation*[Kar89]

This chapter describes the investigation of Cole's parallel merge sort algorithm. It starts by describing why this parallel algorithm is an important contribution to theoretical computer science. The practical value of Cole's algorithm is evaluated by comparing its performance with various simple sorting algorithms. The performance of these simple algorithms are summarised in Section 4.1.

The main part of the chapter is a top down and complete explanation of Cole's algorithm, followed by a description of how it has been implemented on the CREW PRAM simulator. This includes details that are not available in earlier documentation of the algorithm. The chapter ends by presenting the results from the comparison of Cole's algorithm with the simpler algorithms.

4.1 Parallel Sorting

"As an aside, it is interesting to speculate on what are the all time most important algorithms. Surely the arithmetic operations $+$, $-$, $*$, and $\div$ on decimal numbers are basic. After that, I suggest fast sorting and searching, ... "

Stephen A. Cook in his Turing Award Lecture *An Overview of Computational Complexity* [Coo83].

4.1.1 Parallel Sorting—The Continuing Search for Faster Algorithms

The literature on parallel sorting is very rich; in 1986, there was published a bibliography containing nearly four hundred references [Ric86]. Nevertheless, it is still appearing new interesting parallel sorting algorithms.

Within theoretical computer science there are mainly two computational models that have been considered for parallel sorting algorithms—the circuit model and the PRAM model. An early and important result for the circuit model was the odd-even merge and bitonic merge sorting networks presented by Batcher in 1968 [Bat68]. A bitonic merge sort network sorts n numbers in $O(\log^2 n)$ time, see for instance [Akl85].

The AKS sorting network

The first parallel sorting algorithm using only $O(\log n)$ time was presented by *Ajtai, Komlós and Szemerédi* in 1983 [AKS83]. This algorithm is often called the three Hungarians's algorithm, or the AKS-network. The original variant of the AKS-network used $O(n \log n)$ processors and was therefore not cost-optimal (recall Definition 2.18 at page 47). However, Leighton showed in 1984 that the AKS-network can be combined with a variation of the odd-even network to give an optimal sorting network with $O(\log n)$ time on $O(n)$ processors [Lei84]. Leighton points out that the constant of proportionality of this algorithm is immense and that other parallel sorting algorithms will be faster as long as $n < 10^{100}$, a problem size that probably never will occur in practice![1]

[1]The total number of elementary particles in the observable part of the universe has been estimated to about 10^{80} [Sag81].

In spite of being commonly thought of as a purely theoretical achievement, the AKS-network was a major breakthrough; it proved the possibility of sorting in $O(\log n)$ time, and *implied* the first cost optimal parallel sorting algorithm described by Leighton. The optimal *asymptotical* behaviour initiated a search for improvements for closing the gap between its theoretical importance and its practical use. One such improvement is the simplification of the AKS-network done by Paterson [Pat87]. However, the algorithm is still very complicated and the complexity constants remain impractically large [GR88].

Cole's CREW PRAM sorting algorithm

The PRAM model is more powerful than the circuit model. Even the weakest of the PRAM models, the EREW PRAM, may implement a sorting circuit such as Batcher's network without loss of efficiency.

Also for the PRAM model, there has been a search for a $O(\log n)$ time parallel sorting algorithm with optimal cost. In 1986, *Richard Cole* presented a new parallel sorting algorithm called *parallel merge sort* with this complexity [Col86]. This was an important contribution, since Cole's algorithm is the second $O(\log n)$ time and $O(n)$ processor sorting algorithm—the first was the one implied by the AKS-network. Further, it is claimed to have complexity constants which are much smaller than that of the AKS-network. To make it possible to evaluate its practical value, it was necessary to develop an implementation of the algorithm. This work is described in Section 4.2 and Section 4.3.

Are algorithms using more than n processors practical?

Many researchers would argue that sorting algorithms requiring more than n processors to sort n items are of little practical interest. A reasonable, general assumption is that the number of processors, p, should be smaller than n. However, there are situations where $p > n$ should be considered.

Imagine that you are given the task of making an extremely fast sorting algorithm for up to 16 000 numbers on a Connection Machine with 64 000 processor. Reading the description of Cole's $O(\log n)$ time algorithm in the book on "efficient" parallel algorithms by Gibbons and Rytter [GR88], it is far from evident that Cole's algorithm should *not* be considered.

It is interesting in this context to read the recent work of Valiant on the BSP model [Val90] (see Section 2.1.3.4), where he advocates the need for algorithms with parallel slackness. Only the future can tell us whether

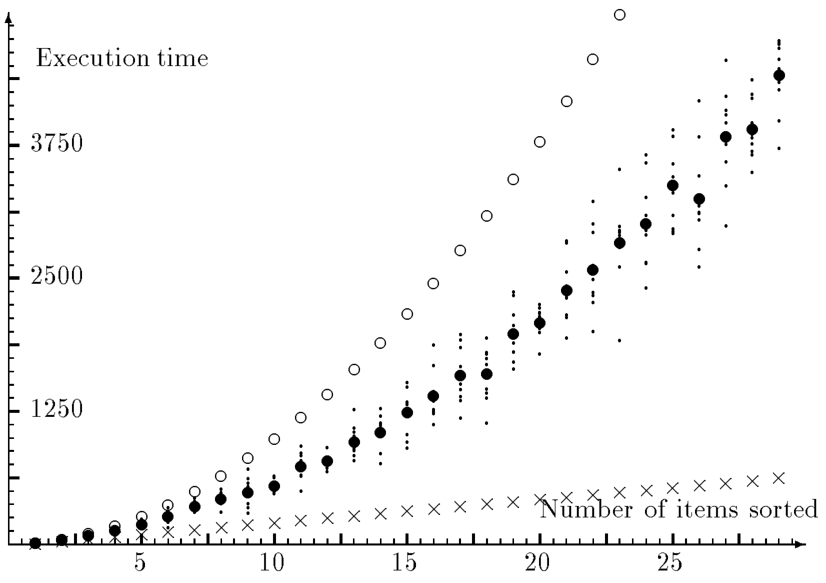


Figure 4.1: Insertion sort algorithm [Ben86] which demonstrates worst, average and best case behaviour. $\times$ represents *best case*—which for this algorithm is a sorted sequence. $\circ$ represents the *worst case*, *i.e.* an input sequence sorted in the opposite order. Each small dot ($\cdot$) shows the time used to sort one of ten random input sequences for each problem size (x-axis), and $\bullet$ shows the *average of these ten samples*. The random cases were generated by drawing numbers from an uniform distribution.

algorithms with very high (virtual) processor requirement will be widely used.

4.1.2 Some Simple Sorting Algorithms

This section summarises the performance of the three simple sorting algorithms described earlier in Chapter 3. For all sorting algorithms evaluated in this thesis, the execution time is measured from the algorithm starts with the input sequence placed in the global memory, until it finishes with the sorted sequence placed in the global memory. The time is measured in number of CREW PRAM time units.

4.1.2.1 Insertion Sort

Perhaps the simplest sequential sorting algorithm is the straight insertion sort algorithm outlined in Section 3.3.2.2, see Figure 3.16 at page 88. A

Table 4.1: Performance data for a CREW PRAM implementation of the parallel insertion sort algorithm presented in Figure 3.17 at page 89. Time and cost are given in *kilo* CREW PRAM time units and *kilo* CREW PRAM (unit-time) instructions respectively. The number of read/write operations to/from the global memory are given in *kilo* locations.

Problem size n	time	#processors	cost	#reads	#writes
4	0.4	3	1.3	0.2	0.06
8	1.1	7	7.8	1.8	0.3
16	2.8	15	41.4	7.9	1.5
32	6.5	31	203.2	36.7	7.5
64	15.2	63	954.5	175.9	34.4
128	34.3	127	4361.6	817.2	154.3
256	76.7	255	19557.0	3118.5	711.8

tuned version (the second program from the top of page 108 in [Ben86]) has been implemented on the CREW PRAM simulator. The execution time of the algorithm is strongly dependent on the degree of presortedness in the problem instance. This is illustrated in Figure 4.1.

4.1.2.2 Parallel Insertion Sort

The parallel version of the insertion sort algorithm described in Section 3.16 (Figure 3.17 at page 89) has also been implemented on the simulator. Its performance is summarised in Table 4.1.

4.1.2.3 Odd-Even Transposition Sort

Table 4.2 summarises the performance of the implementation of the odd-even transposition sort algorithm using $n/2$ processors, which was developed and described in Section 3.4.

4.1.3 Bitonic Sorting on a CREW PRAM

This section gives a brief description of a CREW PRAM implementation of Batcher's bitonic sorting network and its performance.

Table 4.2: Performance data measured from execution of *OddEvenSort* on the CREW PRAM simulator. Time and cost are given in *kilo* CREW PRAM time units and *kilo* CREW PRAM (unit-time) instructions respectively. The number of read/write operations to/from the global memory are given in *kilo* locations.

Problem size n	time	#processors	cost	#reads	#writes
4	0.2	2	0.3	0.02	0.01
8	0.3	4	1.1	0.08	0.03
16	0.5	8	3.6	0.3	0.1
32	0.8	16	12.7	1.1	0.6
64	1.5	32	46.6	4.2	2.3
128	2.8	64	176.5	16.6	8.0
256	5.3	128	683.7	65.9	34.8
512	10.5	256	2683.9	262.9	130.4
1024	20.7	512	10622.5	1050.1	502.5

4.1.3.1 Algorithm Description and Performance

Batcher's bitonic sorting network [Bat68] for sorting of $n = 2^m$ items consists of $\frac{1}{2}m(m+1)$ columns each containing $n/2$ comparators (comparison elements). A detailed description of the algorithm may be found in Chapter 2 of Akl's book on parallel sorting [Akl85]. In this section we assume that n is a power of 2. A natural emulation on a CREW PRAM is to use $n/2$ processors which are dynamically allocated to the one active column of comparators as it moves from the input side to the output side through the network.[2] The global memory is used to store the sequence when the computation proceeds from one step (*i.e.* comparator column) to the next. The main program and its time consumption are shown in Figure 4.2 and Table 4.3.

When discussing time consumption of PPP programs, we will in general use the following notation.:

Definition 4.1 $t(i,n)$ denotes the time used on *one single* execution of statement i of the discussed program when the problem size is n. $t(j..k,n)$ is a short hand notation for $\sum_{i=j}^{i=k} t(i,n)$. $\square$

[2]The possibility of sorting several sequences simultaneously in the network by use of pipelining is sacrificed by this method. This is not relevant in this comparison, since Cole's algorithm does not have a similar possibility.

```
    CREW PRAM procedure BitonicSort
    begin
(1)     assign $n/2$ processors;
(2)     for each processor do begin
(3)         Initiate processors;
(4)         for $Stage := 1$ to $\log n$ do
(5)             for $Step := 1$ to $Stage$ do begin
(6)                 EmulateNetwork;
(7)                 ActAsComparator;
                end;
        end;
    end;
```

Figure 4.2: Main program of a CREW PRAM implementation of Batcher's bitonic sorting algorithm.

Table 4.3: Time consumption of *BitonicSort*.

$t(1, n)$	$=$	$42 + 23\lfloor\log(n/2)\rfloor$
$t(2..3, n)$	$=$	38
$t(4, n)$	$=$	10
$t(5..7, n)$	$=$	84

Table 4.4: Performance data measured from execution of *BitonicSort* on the CREW PRAM simulator. Time and cost are given in *kilo* CREW PRAM time units and *kilo* CREW PRAM (unit-time) instructions respectively. The number of read/write operations to/from the global memory are given in *kilo* locations.

Problem size n	time	#processors	cost	#reads	#writes
4	0.4	2	0.7	0.03	0.02
8	0.6	4	2.6	0.07	0.06
16	1.0	8	8.2	0.2	0.2
32	1.5	16	23.7	0.6	0.5
64	2.0	32	64.6	1.5	1.4
128	2.6	64	169.0	3.9	3.7
256	3.4	128	428.2	9.9	9.4
512	4.1	256	1058.3	24.3	23.3
1024	5.0	512	2563.6	58.8	56.8
2048	6.0	1024	6107.1	140.3	136.2
4096	7.0	2048	14346.2	329.7	321.5

EmulateNetwork is a procedure which computes the addresses in the global memory corresponding to the two input lines for that comparator in the current *Stage* and *Step*. *ActAsComparator* calculates which of the two possible comparator functions that should be done by the processor (comparator) in the current *Stage* and *Step*, performs the function, and writes the two outputs to the global memory. Both procedures are easily performed in constant time.

The main program shown in Figure 4.2 implies that the time consumption of the implementation may be expressed as

$$T(\mathit{BitonicSort}, n) = t(1..3, n) + t(4, n) \times \log n + t(5..7, n) \times \frac{1}{2} \log n (\log n + 1) \tag{4.1}$$

Note that this equation gives an *exact* expression for the time consumption for *any* value of n, (where n is a power of 2).

4.2 Cole's Parallel Merge Sort Algorithm — Description

"Cole [Col86] has invented an ingenious version of parallel merge sorting in which this merging is done in $O(1)$ parallel time."
Alan Gibbons and Wojciech Rytter in *"Efficient Parallel Algorithms"* [GR88].

Richard Cole presented two versions of the parallel merge sort algorithm, one for the CREW PRAM model and a more complex version for the EREW PRAM model [Col86]. Throughout this thesis, we are considering the CREW variant,

without the modification presented at the bottom of page 776 in [Col88]. The EREW variant is substantially more complex.

A revised version of the original paper was published in the *SIAM Journal on Computing* in 1988 [Col88]. This paper has been used as the main reference for my study and implementation of the CREW PRAM variant of the algorithm. Cole did also write a technical report about the algorithm [Col87], but has informally expressed that it is less good than the SIAM paper [Col90].

This section explains Cole's CREW PRAM parallel merge sort algorithm. Section 4.3 describes the most important decisions that were made during the implementation. This is followed by pseudo code for the main parts of the algorithm along with an *exact* analysis of the time consumption.

Motivation for the Description

Many researchers in the theory community refer to Cole's algorithm as *simple*. However, the effort needed to get the level of understanding that is required for implementing the algorithm is probably counted in days and not hours for most of us. It is therefore natural to use a top down approach in describing the algorithm. The reader may consult the SIAM paper [Col88] for some further details or an alternative presentation. The description found at pages 188–198 in the book *Efficient Parallel Algorithms* by Gibbons and Rytter [GR88] is not completely consistent with Cole's paper, but may add some understanding.

4.2.1 Main Principles

Cole's parallel merge sort assumes *n distinct items*. These items are distributed one per leaf in a complete binary tree—thus it is assumed that *n is a power of 2*. At each node in the tree, the items in the left and right subtree of that node are merged. The merging starts at the leaves and proceeds towards the top of the tree. Before the merging in a node is completed, samples of the items received so far are sent further up the tree. This makes it possible to merge at several levels of the tree simultaneously in a pipelined fashion.

As a motivation, let us consider two other parallel sorting algorithms based on a n-leaf complete binary tree.

4.2.1.1 Two Simpler Ways of Merging In a Binary Tree

One processor in each node
Perhaps the simplest way to sort n items initially distributed on the n leaf nodes of a binary tree, is to assign one processor to each node, and to merge subsequences level by level starting at the bottom of the tree: At the first stage, the nodes one level above the leaf nodes will merge two sequences of length 1 into one sequence of length 2. In the next stage, the nodes one level higher up will merge two sequences into one sequence of length 4, and so on. The *degree of parallelism is restricted to the number of nodes in the active level* of the tree. Worse though, as the algorithm proceeds towards the top of the tree, the work to be done in each node increases, while the parallelism becomes smaller. At the top node, two sequences of length $n/2$ are merged by one single processor. This last stage alone takes $O(n)$ time, and this form of tree-based merge sort is definitely not "fast".

Parallel processing inside each node
Cole gives a motivating example that is similar to the algorithm just sketched [Col88]. The computation proceeds up the tree, level by level from the leaves to the root. Each node merges the sorted sets computed at its children. But, as Cole points out, merging in a node may be done by the $O(\log\log n)$ time n-processor merging algorithm of Valiant [Val75]. In this case we also have parallelism inside the nodes at the active level. However, the merging is still done *level by level*—resulting in $O(\log n \log\log n)$ time consumption.

4.2.1.2 Cole's Observation: The Merges at the Different Levels of the Tree Can be Pipelined

Borodin and Hopcroft have proved that there is an $\Omega(\log \log n)$ lower bound for merging two sorted arrays of n items using n processors [BH85]. Therefore, as Cole points out, it is not likely that the level by level approach may lead to an $O(\log n)$ time sorting algorithm.

Cole's algorithm is based on an $O(\log n)$ time merging procedure which is *slower* than the merging algorithm of Valiant. The main principle is described by the following simpler, but similar merging procedure;

> *Cole's* $\log n$ *merging procedure:*
> "The problem is to merge two sorted arrays of n items. We proceed in $\log n$ stages. In the ith stage, for each array, we take a sorted sample of 2^{i-1} items, comprising every $n/2^{i-1}$th item in the array. We compute the merge of these two samples." ([Col88] page 771.)

Cole made two key observations:

> *Merging in constant time:*
> Given the results of the merge from the $i-1$st stage, the merge in the ith stage can be done in $O(1)$ time.

Since we have $\log n$ levels in the tree, using this merging procedure in the level by level approach gives an $O(\log^2 n)$ time sorting algorithm.

The second observation explains how the use of a slower merging procedure may result in a faster sorting algorithm:

> *The merges at the different levels of the tree can be pipelined:*
> This is possible since merged *samples* made at level l of the tree may be used to provide samples of the appropriate size for merging at the next level above l without losing the $O(1)$ time merging property.

This may need som further explanation. Consider node u at level l, see Figure 4.3. In the *level by level approach*, as soon as node v (and w) store a sorted sequence containing *all the items* in the subtree rooted at v (and w), the merging may start at level l. The merging in node u takes $\log k$ stages where k is the size of the sorted sequences in node v and w. Each of these stages may be done in constant time due to the systematical way

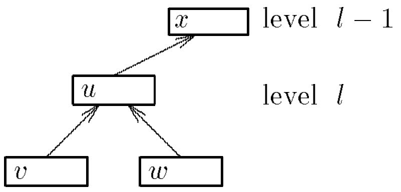


Figure 4.3: An arbitrary node u, its two child nodes v and w, and parent node x. The root of the tree is at level 0.

of sampling the sorted sequences in v and w.[3] When node u has completed the merging, the merging may start at node x.

In the pipelined approach, the nodes are allowed to start much earlier on the merging procedure. Once node u contains 4 items, it starts to send *samples* to its parent node x. (These samples are taken from the sequence stored in node u, which in *this* case may only be a sample of the items initially stored in the leaves of the subtree rooted at u.) Node x follows the same merging procedure—as soon as it contains 4 items it starts sending samples to its parent node, and so on. It is shown in [Col88] that this way of letting the nodes start to merge sorted subsequences from its left and right child node, before these two nodes have finished their merging, may be implemented without destroying the possibility of doing each merge *stage* in constant time.

4.2.2 More Details

This section gives a more detailed description of Cole's parallel merge sort algorithm. It attempts to provide a thorough understanding, by also explaining some details vital for an implementation which are *not* given in Cole's description [Col88].

[3] In stage i, the sample from v and w may be merged in constant time because the array currently stored in node u (which was obtained in stage $i-1$) is a so called *good sampler* for the new samples from v and w. The term "good sampler" is used by Gibbons and Rytter [GR88] and reflects that the old array may be used to split the new array into roughly equal sized parts in the following manner. (Informally) If the items in the new and bigger array are inserted at their right positions (with respect to sorted order) in the old array, a maximum of three items from the new array will be inserted between any two neighbour items in the old array. (Full details in [Col88].)

4.2.2.1 The Task of Each Node u

Each node u should produce a list $L(u)$, which is the sorted list containing the items distributed initially at the leaves of the subtree with u as its top node (root). During the progress of the algorithm, each node u stores an array $\mathrm{Up}(u)$ which is a sorted subset of $L(u)$. Initially, all leaf nodes y have $\mathrm{Up}(y) = L(y)$. At the termination of the algorithm the top node of the whole tree, t, has $\mathrm{Up}(t) = L(t)$ which is the sorted sequence. In the progress of the algorithm, $\mathrm{Up}(u)$ will generally contain a sample of the items in $L(u)$.

Up(u), SampleUp(v), SampleUp(w) and NewUp(u)

Before we proceed, we need some simple definitions. A node u is said to be *external* if $|\mathrm{Up}(u)| = |L(u)|$, otherwise it is an *inside* node. Initially, only the leaf nodes are external. In each stage of the algorithm, a new array, called $\mathrm{NewUp}(u)$ is made in each inside node u. $\mathrm{NewUp}(u)$ is formed in the following manner (see Figure 4.4.)

(Phase 1) Samples from the items in $\mathrm{Up}(v)$ and $\mathrm{Up}(w)$ are made and stored in the arrays $\mathrm{SampleUp}(v)$ and $\mathrm{SampleUp}(w)$, respectively.

(Phase 2) $\mathrm{NewUp}(u)$ is made by merging $\mathrm{SampleUp}(v)$ and $\mathrm{SampleUp}(w)$ *with help* of the array $\mathrm{Up}(u)$.

For all nodes, the $\mathrm{NewUp}(u)$ array made in stage i is the $\mathrm{Up}(u)$ array at the start of stage $i+1$. At external nodes, Phase 2 is not performed—since these nodes have already produced their list $L(u)$.

4.2.2.2 Phase 1: Making Samples

The samples $\mathrm{SampleUp}(u)$ are made in parallel by all nodes u according to the following rules:

1. If u is an *inside* node, $\mathrm{SampleUp}(u)$ is the sorted array comprising of every fourth item in $\mathrm{Up}(u)$. The items are taken from the positions 1, 5, 9, etc.. If $\mathrm{Up}(u)$ contains less than four items, $\mathrm{SampleUp}(u)$ becomes empty.

2. At the first stage when u is external, $\mathrm{SampleUp}(u)$ is made in the same way as for inside nodes. At the second stage as external node, $\mathrm{SampleUp}(u)$ is every second item in $\mathrm{Up}(u)$, and in the third stage, $\mathrm{SampleUp}(u)$ is every item in $\mathrm{Up}(u)$. $\mathrm{SampleUp}(u)$ is always in sorted order.

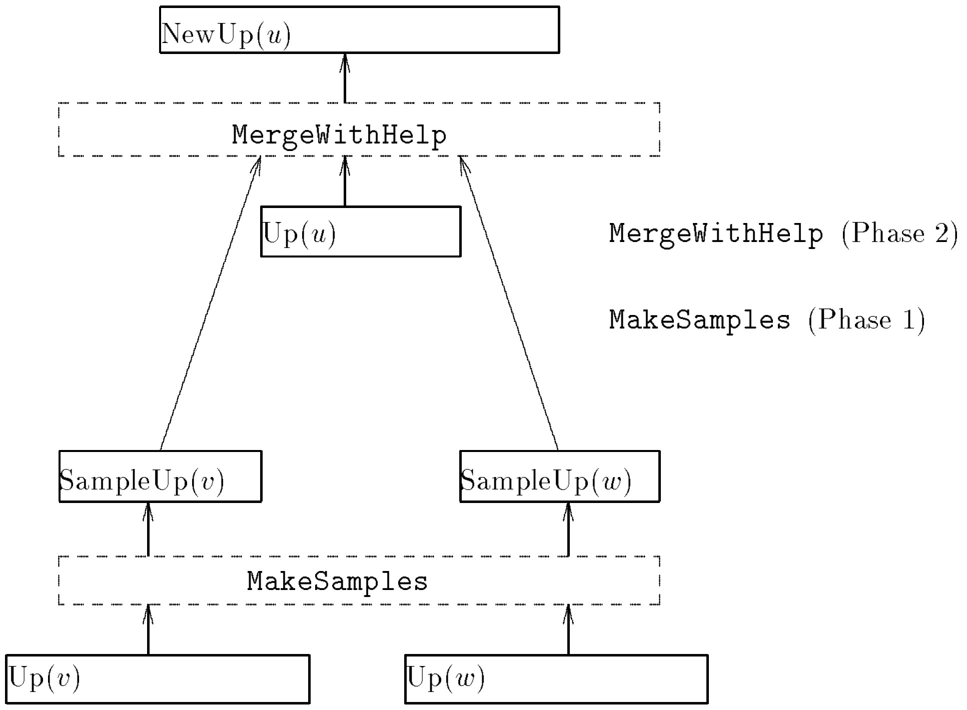


Figure 4.4: Computation of NewUp(u) in two main phases.

The algorithm has $3 \log n$ stages
Initially, only the leaves are external nodes. In the third stage as external, a node v has used all the items in $L(v)$ to produce SampleUp(v) in Phase 1, and its parent node u has merged SampleUp(v) and SampleUp(w) into Up(u) in Phase 2. Node u has now received all the items in $L(u)$, and the child nodes v and w have finished their tasks and may "terminate". In the next stage, the parent node u will have $\text{Up}(u) = L(u)$, and this is the first stage as external for node u.

We see that the level with external nodes moves up one level every third stage. After $3 \log n$ stages the top node t becomes external, it has $\text{Up}(t) = L(t)$, and the algorithm may stop (Lemma 2 in [Col88]).

The size of Up(u) is doubled in each stage
The size of the samples made by the external nodes double in each stage, because the sample rate is halved in each stage. As a consequence, the size of Up(u) in the inside nodes one level higher does also double in each stage. At this level, the sample rate is fixed (4) as long as the nodes are inside nodes. But, since $|\text{Up}(u)|$ doubles in each stage, the samples made at this level will also double in each stage. By induction, we see intuitively that the size of Up(u) doubles in each stage for each inside node with $|\text{Up}(u)| > 0$.

4.2.2.3 Phase 2: Merging In $O(1)$ Time

The most complicated part of Cole's algorithm is the merging of SampleUp(v) and SampleUp(w) into NewUp(u) in $O(1)$ time. It constitutes the major part of the algorithm description in [Col88], and about 90% of the code in the implementation. The merging is described in the following. Proofs and some more details are found in [Col88].

Using ranks to compute NewUp(u)
The merging is based on maintaining a set of *ranks*. Assume that A and B are sorted arrays. Informally we define A to be ranked in B (denoted $A \rightarrow B$) if we for each item e in A know how many items in B that are smaller or equal to e. Knowing $A \rightarrow B$ means that we know where to insert an item from A in B to keep B sorted.

When using this concept in an implementation, it is necessary to do the following distinction: The rank of an item x in a sorted array S is denoted $\text{R}(x, S)$. When using $\text{R}(x, S)$ to represent the correct position (with respect to) of x in S—it is important to know whether x is in S or not, as made clear in the following observation.

Observation 4.1
Assume that the positions of the items in the sorted array S are numbered $\{1, 2, 3, \ldots |S|\}$. If $x \in S$, $\mathrm{R}(x, S)$ represents the correct position of x in S. However, if $x \notin S$, $\mathrm{R}(x, S)$ gives the number of items in S which are smaller than x, thus if x is to be inserted in S, its correct position will be $\mathrm{R}(x, S)+1$.

At the start of each stage we know Up(u), SampleUp(v) and SampleUp(w). In addition, we do the following assumption;

Assumption 4.1
At the start of each stage, for each inside node u with child nodes v and w we know the ranks Up(u) $\rightarrow$ SampleUp(v), and Up(u) $\rightarrow$ SampleUp(w).

The making of NewUp(u) may be split into two main steps; the merging of the samples from its two child nodes, and the maintaining of Assumption 4.1. The calculation of the ranks constitutes a large fraction of the total time consumption of Cole's algorithm. However, as we will see, they are crucial for making it possible to merge in constant time.

4.2.2.4 Phase 2, Step 1: Merging

We want to merge SampleUp(v) and SampleUp(w) into NewUp(u), see Figure 4.5. This is easy if we for each item in SampleUp(v) and SampleUp(w) know its position in NewUp(u). The merging is then done by simply writing each item to its right position.

Consider an arbitrary item e in SampleUp(v), see Figure 4.5. We want to compute the position of e in NewUp(u), *i.e.*, the rank of e in NewUp(u), denoted R(e, NewUp(u)). Since NewUp(u) is made by merging SampleUp(v) and SampleUp(w) we have

$$\mathrm{R}(e, \mathrm{NewUp}(u)) = \mathrm{R}(e, \mathrm{SampleUp}(v)) + \mathrm{R}(e, \mathrm{SampleUp}(w)) \qquad (4.2)$$

The problem of merging has been reduced to computation of R(e, SampleUp(v)) and R(e, SampleUp(w)). Since e is from SampleUp(v), R(e, SampleUp(v)) is just the position of e in SampleUp(v). Computing the rank of e in SampleUp(w) is much more complicated. Similarly, for items e in SampleUp(w) it is easy to compute R(e, SampleUp(w)), but difficult to compute R(e, SampleUp(v)).

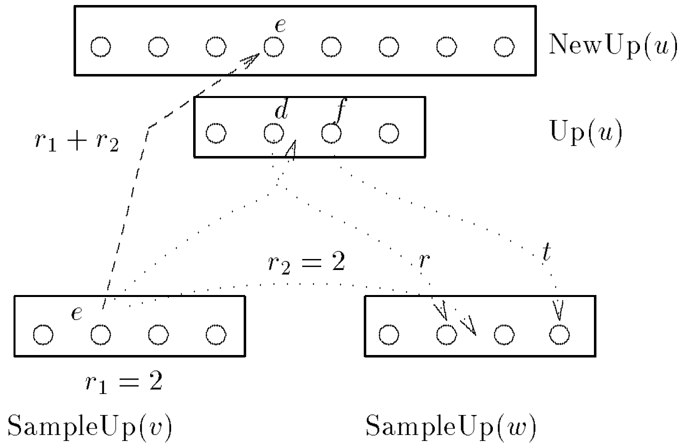


Figure 4.5: The use of *ranks* in Cole's $O(1)$ time merging procedure. Knowing the position of the item e in SampleUp(v), R(e, SampleUp(v)) = r_1, and its position in SampleUp(w), R(e, SampleUp(w)) = r_2, the position of e in NewUp(u), R(e, NewUp(u)) is simply $r_1 + r_2$ (4 in this example).

The computation of the ranks SampleUp(v) $\rightarrow$ SampleUp(w) and SampleUp(w) $\rightarrow$ SampleUp(v) is termed computation of *crossranks*. In the mass of details and levels which follows, Figure 4.6 may be used as an aid for keeping track of "where we are" in the algorithm.

4.2.2.5 Computing Crossranks

Up(u) and Assumption 4.1 may be to a great help in computing the crossranks. Consider Figure 4.5 and the item e from SampleUp(v). We want to find the position of e in SampleUp(w) as if it was to be inserted according to sorted order in that array.

Note that it is only the situation where e is from SampleUp(v) that is described in this and the following sections (4.2.2.6 and 4.2.2.7). The other case, e is from SampleUp(w), is handled in a completely symmetric manner.

First, for each item e in SampleUp(v) we compute its rank in Up(u). This computation is done for all the items in SampleUp(v) in parallel, and is described as *Substep 1* in Section 4.2.2.6 below.

R(e, Up(u)) gives us the items d and f in Up(u) which would *straddle*[4] e

[4]Let x, y and z be three items, with $x < z$. We say that x and z *straddle* y if $x \leq y$

- One stage in Cole's parallel merge sort algorithm
  - Phase 1: Making Samples (page 115)
  - Phase 2: Merging in $O(1)$ time (page 117)
    - Step 1: Merging (page 118)
      - Computing crossranks (page 119)
        - Substep 1 (page 121)
        - Substep 2 (page 122)
      - Make NewUp(u) (Eq. 4.2 at page 118)
    - Step 2: Maintaining Ranks (page 125)
      - ...from sender node (page 125)
      - ...from other node (page 126)

Figure 4.6: Main structure of one stage in Cole's algorithm.

if e had to be inserted in Up(u). Further, Assumption 4.1 gives R(d, SampleUp(w)) and R(f, SampleUp(w))—the right positions for inserting d and f in SampleUp(w). These ranks are called r and t in Figure 4.5. Informally, since $e \in [d, f\rangle$ the right position for e in SampleUp(w) is bounded by the positions (ranks) r and t. It can be shown that r and t always specifies a range of maximum 4 positions, and the right position, R(e, SampleUp(w)), may thus be found in constant time. This is explained in more detail as *Substep 2* in Section 4.2.2.7.

Before the two substeps are explained in more detail, we mention an analogy which may help to one's understanding: Imagine that you are supposed to navigate to a certain position (R(e, SampleUp(w))) far away in an unknown area (SampleUp(w)). You know where you are, and have a detailed map covering the entire area. The normal smart way to solve such a problem is to split it into two succeeding steps; coarse and fine navigation.

Coarse navigation is done by using an imagined high-level map where many details have been omitted. This map (Up(u)) is simpler than the full detail map (SampleUp(w)), but it is good enough (good sampler) for navigating to a small area (given by r and t) containing the exact position. *Fine navigation* is then done by detailed searching in this small area (comparing e with the elements given by r and t).

and $y < z$, *i.e.*, $y \in [x, z\rangle$.

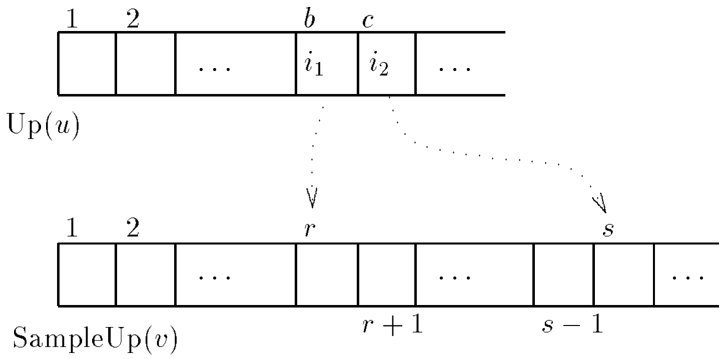


Figure 4.7: Substep 1: Computing SampleUp(v) $\rightarrow$ Up(u) by using Up(u) $\rightarrow$ SampleUp(v)

4.2.2.6 Substep 1: Computing SampleUp(v) $\rightarrow$ Up(u)

SampleUp(v) $\rightarrow$ Up(u) is computed by using Up(u) $\rightarrow$ SampleUp(v). Consider an arbitrary item i_1 in Up(u), see Figure 4.7. The range $[i_1, i_2\rangle$ is defined to be the interval *induced* by item i_1. We want to find the set of items in SampleUp(v) which are contained in $[i_1, i_2\rangle$, denoted $I(i_1)$.

The items in $I(i_1)$ have rank in Up(u) equal to b, where b is the position of i_1 in Up(u), and the array positions are numbered starting with 1 as shown in the figure. Once $I(i_1)$ have been found, a processor associated with item i_1 may assign the rank b to the items in the set. Simultaneously, a processor associated with i_2 should assign the rank c to $I(i_2)$.

Computing $I(i_1)$

The precise calculations necessary for finding $I(i_1)$ are not explained in Cole's SIAM paper [Col88]. However, we must understand how it can be done to be able to implement the algorithm. We want to find all items $\alpha \in$ SampleUp(v) with R(α, Up(u)) $= b$. These items must satisfy

$$(\alpha \geq i_1) \wedge (\alpha < i_2) \tag{4.3}$$

Let item(j) denote the item stored in position j of SampleUp(v). Assumption 4.1 gives R(i_1, SampleUp(v)) $= r$ and R(i_2, SampleUp(v)) $= s$, which implies

$$(\text{item}(r) \leq i_1) \wedge (\text{item}(s) \leq i_2) \tag{4.4}$$

We have the following observation

Observation 4.2
(See Figure 4.7.) If there exist items β in SampleUp(v) between (and not including) position r and s they must have R(β, Up(u)) $= b$.

Proof: Let item$(r+1)$ denote an item which is to the right of item(r) and to the left of item(s), see Figure 4.7. The definition of the rank r implies that $i_1 <$ item$(r+1)$. Hence, R(item$(r+1)$, Up$(u)) \geq b$. Let item$(s-1)$ denote an item which is to the left of item(s) and to the right of item(r). The requirement of distinct items implies item$(s) >$ item$(s-1)$, and the definition of the rank s implies item$(s) \leq i_2$. Distinct items then gives item$(s-1) < i_2$, so we have R(item$(s-1)$, Up$(u)) < c$. Since $b = c-1$, all items in SampleUp(v) in positions $r+1, r+2, \ldots, s-1$ have rank b in Up(u). □
The items in positions r and s must be given special treatment, we have:

$$\begin{aligned}
&\text{item}(r) = i_1 \Rightarrow \text{R}(\text{item}(r), \text{Up}(u)) = b\\
&\text{item}(r) < i_1 \Rightarrow \text{R}(\text{item}(r), \text{Up}(u)) < b\\
&\text{item}(s) = i_2 \Rightarrow \text{R}(\text{item}(s), \text{Up}(u)) = c\\
&\text{item}(s) < i_2 \Rightarrow \text{R}(\text{item}(s), \text{Up}(u)) < c \Rightarrow \text{R}(\text{item}(s), \text{Up}(u)) = b
\end{aligned}$$

To conclude, $I(i_1)$ is found by comparing item(r) with i_1 and item(s) with i_2. A processor associated with item i_1 may do these simple calculations and assign the rank b to all items in $I(i_1)$ in *constant* time. This is possible because it can be proved that $I(y)$ for any item y in Up(u) will contain at most three items.[5]

4.2.2.7 Substep 2: Computing SampleUp(v) → SampleUp(w)

See Figure 4.8. As described in Section 4.2.2.5, R(e, Up(u)) (which was computed in Substep1) gives the straddling items d and f in Up(u). Further, Assumption 4.1 gives the ranks r and t. We want the *exact* position of e if it had to be inserted in SampleUp(w). *Which* items in SampleUp(w) must be compared with e? The question is answered by the following two observations.

[5]Cole proves that Up(u) is a so called *3-cover* for SampleUp(v) [Col88]. 3-cover is the same concept as *good sampler* mentioned in Footnote 3 at page 114. Up(u) is a 3-cover for SampleUp(v) if each interval $[i_1, i_2\rangle$ *induced* by an item i_1 from Up(u) contains at most 3 items from SampleUp(v), see Figure 4.7. Note that the definition allows $i_1 = -\infty$ as an imagined item to the left of the first element of Up(u), and similarly $i_2 = \infty$.

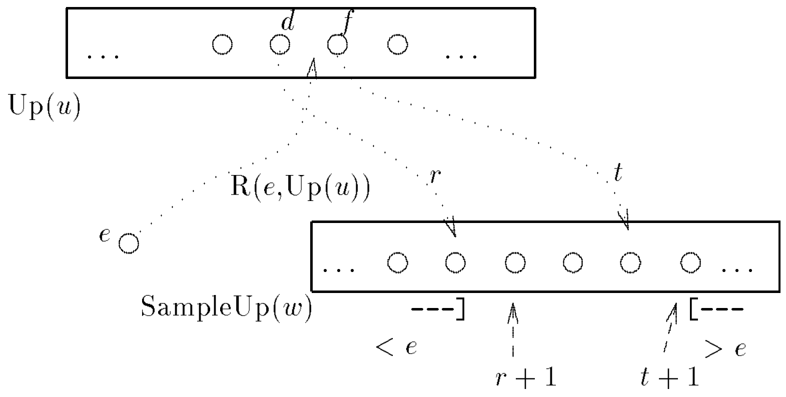


Figure 4.8: Substep 2: Computing SampleUp(v) $\rightarrow$ SampleUp(w) by using SampleUp(v) $\rightarrow$ Up(u)

Observation 4.3
All items in SampleUp(w) to the left of, and including, position r are smaller than item e.

Proof: Let item(x) denote the item in SampleUp(w) at position x. Since d and f straddle e, we have $e \geq d$. The rank r implies that $d \geq$ item(r), so we have $e \geq$ item(r). The assumption of distinct elements implies that SampleUp(v) and SampleUp(w) never will contain the same element, hence $e \neq$ item(r). This implies that $e >$ item((r)). $\square$

Observation 4.4
All items in SampleUp(w) to the right of position t are larger than item e.

Proof: The rank t implies that $f <$ item($t+1$), and we know that $e < f$. Therefore, $e <$ item($t+1$). $\square$

How many items must be compared with e? Observations 4.3 and 4.4 tell us that e must be compared with the items from SampleUp(w) with positions in the range $[r+1, t]$. Since Up(u) is a 3-cover for SampleUp(w) we know that $[d, f\rangle$ contains at most three items in SampleUp(w). But, the set of items in SampleUp(w) contained in $[d, f\rangle$ is not *necessarily* the same as the items with positions in the range $[r+1,t]$. However, we can prove the following observation by using the 3-cover property of $[d,f\rangle$.

Observation 4.5
(See Figure 4.8). Item e must be compared with *at most three items* from SampleUp(w) starting with item($r+1$), going to the right, but not beyond item(t). In other words, the items item($r+i$) for $1 \le i \le \min(3, t-r)$.

Proof: First, let us consider which items in SampleUp(w) may be contained in $[d,f\rangle$. We have four possible cases;

$$\begin{array}{ll} 1) & d = \text{item}(r) \Rightarrow \text{item}(r) \in [d,f\rangle \\ 2) & d \ne \text{item}(r) \Rightarrow d > \text{item}(r) \Rightarrow \text{item}(r) \notin [d,f\rangle \\ 3) & f = \text{item}(t) \Rightarrow \text{item}(t) \notin [d,f\rangle \\ 4) & f \ne \text{item}(t) \Rightarrow f > \text{item}(t) \Rightarrow \text{item}(t) \in [d,f\rangle \end{array} \tag{4.5}$$

which imply four cases for the "placement" of $[d,f\rangle$ in SampleUp(w);

$$\begin{array}{ll} i) & [\text{item}(r), \text{item}(t-1)] \\ ii) & [\text{item}(r), \text{item}(t)] \\ iii) & [\text{item}(r+1), \text{item}(t-1)] \\ iv) & [\text{item}(r+1), \text{item}(t)] \end{array}$$

Case *i)*: The 3-cover property implies that $|[\text{item}(r), \text{item}(t-1)]| \le 3$. By deleting one and adding one item we get $|[\text{item}(r+1), \text{item}(t)]| \le 3$.
Case *ii)*: The 3-cover property gives $|[\text{item}(r), \text{item}(t)]| \le 3$, which implies $|[\text{item}(r+1), \text{item}(t)]| \le 2$.[6]
Case *iii)*: $|[\text{item}(r+1), \text{item}(t-1)]| \le 3$. At first sight, one may think that this case makes it necessary to compare e with 4 items in SampleUp(w). However, this case does only occur when item(t) $= f$ (see case 3 in equation 4.5), and since $e < f$ we know that $e <$ item(t), so there is no need to consider item(t). We need only consider items in $[\text{item}(r+1), \text{item}(t-1)]$ which contains at most 3 items.
Case *iv)*: The 3-cover property implies $|[\text{item}(r+1), \text{item}(t)]| \le 3$.
Thus we have shown that e must be compared with at most three items for all possible cases. □

Since these (at most) three items in SampleUp(w) are sorted, two comparisons are sufficient to locate the correct position of e. Therefore, R(e, SampleUp(w)) can be computed in $O(1)$ time.

[6]In this case Figure 4.8 is not correct; at most one item (item($r+1$)) may exist between item(r) and item(t).

When Substep 2 has been done (in parallel) for each item e in SampleUp(v) and SampleUp(w), every item knows its position (rank) in NewUp(u) by Equation 4.2, and may write itself to the correct position of that array.

4.2.2.8 Phase 2, Step 2 — Maintaining Ranks

This is only half the story. We assumed in Assumption 4.1 at page 118 that the ranks Up(u) $\rightarrow$ SampleUp(v) and Up(u) $\rightarrow$ SampleUp(w) are available at the start of each stage. We therefore must compute NewUp(u) $\rightarrow$ NewSampleUp(v) and NewUp(u) $\rightarrow$ NewSampleUp(w) at the end of each stage, so that the assumption is valid at the start of the next stage. (NewSampleUp(x) is the SampleUp(x) that will be produced in the *next* stage.)

Doing this in $O(1)$ time is about just as complicated as the merging described above. We explain the computation of NewUp(u) $\rightarrow$ NewSampleUp(v). NewUp(u) $\rightarrow$ NewSampleUp(w) is computed in an analogous way.

Consider an item e in NewUp(u). The computation of R(e, NewSampleUp(v)) is split into two cases. The first case occurs when e came from SampleUp(v) (or e came from SampleUp(w) during the computing of R(e, NewSampleUp(w))). We term this case computation of rank in NewSampleUp *from sender node*, and it is described in Section 4.2.2.9 below. The other case is called computation of rank in NewSampleUp *from other node*, and is described in Section 4.2.2.10.

First we solve the *from sender node* case for all items in NewUp(u). The results of this computation are then used to solve the *from other node* case.

4.2.2.9 Computing the Rank in NewSampleUp From Sender Node

We know Up(u) $\rightarrow$ SampleUp(v) and Up(u) $\rightarrow$ SampleUp(w). NewUp(u) contains exactly the items in SampleUp(v) and SampleUp(w). Therefore, the rank R(α, NewUp(u)) for all items α in Up(u) is easily computed as the sum of R(α, SampleUp(v)) and R(α, SampleUp(w)). This is done for all nodes in parallel.

SampleUp(v) is made from Up(v) by taking every fourth item in that array.[7] Thus, knowing the position of an item in SampleUp(v), it is easy to find the position of the corresponding (same) item in Up(v). Since we

[7]Every fourth item is the normal case. An implementation must also handle the two other sampling rates which may occur when node v is external. (See Section 4.2.2.2.)

just have computed Up(v) $\rightarrow$ NewUp(v) we may compute SampleUp(v) $\rightarrow$ NewUp(v) by going through Up(v).

For each item β in SampleUp(v) we have found R(β, NewUp(v)). Similarly as for SampleUp(v), NewSampleUp(v) is made by taking every fourth item from NewUp(v).[8] Informally, R(β, NewSampleUp(v)) is therefore simply R(β, NewUp(v)) divided by four.

At this point we know SampleUp(v) $\rightarrow$ NewSampleUp(v) *and* that the item e in NewUp(u) came from SampleUp(v). To find R(e, NewSampleUp(v)) it remains to locate the corresponding (same) item e in SampleUp(v). This may easily be done if we for each item in NewUp(u) records the sender address (position) of e in SampleUp(v).

4.2.2.10 Computing the Rank in NewSampleUp From Other Node

See Figure 4.9 which illustrates the case when e is from SampleUp(w). We want to find the correct position of item e if it had to be inserted in NewSampleUp(v). The following assumption will help us.

Assumption 4.2
(See Figure 4.9.) In each stage, at the start of step 2 (Maintaining Ranks), for each item e in NewUp(u) that came from SampleUp(w) we know the straddling items d and f from SampleUp(v). (And vice versa for items from SampleUp(v).)

The item d is the first item in NewUp(u) from "the other node" (SampleUp(v)) to the left of e, and similarly f is the first item to the right.

Further, we know the ranks r and t of d and f in NewSampleUp(v)—they were computed in the *from sender node* case. Compare this situation with *Substep 2* described in Section 4.2.2.7. We see that Figure 4.9 and Figure 4.8 illustrate the same method for computing the correct position of e in NewSampleUp(v) and SampleUp(w) respectively. R(e, NewSampleUp(v)) is found by comparing item e with at most three items in NewSampleUp(v). (Recall Observation 4.5 at page 124.) Since these three items are sorted, two comparisons will suffice.

It remains to explain how to make Assumption 4.2 valid. Consider Figure 4.8 at page 123 which illustrates the calculation of R(e, SampleUp(w)) for an item e in SampleUp(v). (Do *not* consider the items d and f in that figure. They have a slightly different meaning than d and f in Assumption 4.2.)

[8]The same precautions as mentioned in the previous footnote must be taken here. In addition, we must remember to use the sampling rate that will be used in the *next* stage.

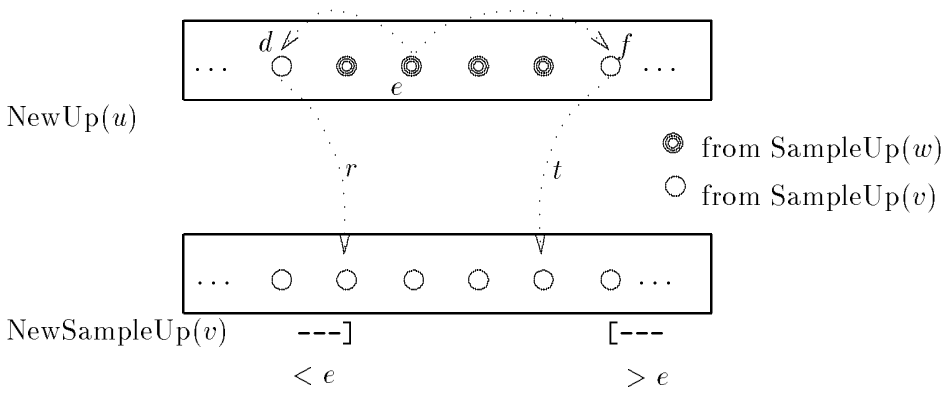


Figure 4.9: Maintaining Ranks: The so called "from other node" case. The straddling items d and f stored at end of Step 1 are used to compute the rank of e in NewSampleUp from the other node. The position of item e in NewSampleUp(v) is bounded by the ranks r and t which are computed in the "from sender node" case.

Since we know $e \notin$ SampleUp(w)(still Figure 4.8), R(e, SampleUp(w)) gives exactly the two items from the other set that straddle e. For an item e from SampleUp(w) a completely symmetric calculation gives R(e, SampleUp(v)), and therefore implicitly the two items d and f from SampleUp(v) that straddle e. These are the items d and f needed in Figure 4.9. Thus, Assumption 4.2 is kept valid if we at the end of Step 1, the making of NewUp(u), record the positions (ranks) of d and f in NewUp(u).

4.3 Cole's Parallel Merge Sort Algorithm, Implementation

This section describes those aspects of the implementation of Cole's parallel merge sort algorithm that are most relevant to parallel processing. Processor activation, how the processors are used in parallel, the communication of local variables between the processors, and CREW PRAM programming are emphasised. Programming details that are well known from sequential programming are given less attention. The text assumes a reasonable knowledge of the algorithm description given in the preceeding section.

4.3.1 Dynamic Processor Allocation

A key to the implementation of Cole's algorithm is to understand how the processors should be allocated to the work which has to be done in the various nodes of the tree during the computation.

Cole's description [Col88] omits many details concerning how the work is distributed on the processors. Program designers may regard this lack of detail as a deficiency—but they should not fail to see its advantages. Cole's description is *detailed enough* to explain the main principles of the processor usage. Still, it is at an *sufficiently high level* for making the programmer free to decide on the details of how the processors are used.

The main principles of the processor usage are described in this section. The full details of the processor usage in the actual implementation are explained in Section 4.3.2.

4.3.1.1 A growing number of active levels

When designing the processor allocation, it is natural to start by finding out where and when there is work to do. As described in Section 4.2.2.2 at page 117, at the end of *every third stage the lowest active level moves one level up towards the top.* Further, the first samples received by an *inside* node u will have size equal to one. Thus $|\text{Up}(u)| = 2$ in the first stage when node u is active. Node u will not produce samples during this stage because $\text{Up}(u)$ is too small. However, in the next stage, the size of $\text{Up}(u)$ will have been doubled[9] and node u will produce samples for its parent node. We see that *the highest active level will move one level upwards every second stage.*

Because of the difference in "speed" of the lowest and highest active level in the tree, the total number of active levels will increase during the computation, until the top level has been reached. This is illustrated in Figure 4.10 for the case $n = 16$.

4.3.1.2 The Number of Items in the Arrays

Cole gives a detailed analysis of the number of items in the Up, NewUp, and SampleUp arrays. The size of the arrays will be different from stage to stage. Consider a node u with $|\text{Up}(u)| > 0$ and child nodes v and w which are not external. Since $|\text{Up}(u)|$ doubles in each stage we have $2|\text{Up}(u)| =$

[9]As explained in Section 4.2.2.2 at page 117.

```
                 ! Stage No:
          Level:! 1  2  3  4  5  6  7  8  9 10 11 12
  root node   0 !                          M  M  M  M
              1 !                    M  M  M  1  2  3
              2 !              M  M  1  2  3
              3 !        M  1  2  3
 leaf nodes   4 ! 1  2  3
```

Figure 4.10: The active levels of the binary tree used by Cole's algorithm for the case $n = 16$. `M` represents that inside nodes at that level perform merging in the stage. i, $(i = 1, 2, 3)$, represents that nodes at that level are in their i'th stage as external nodes.

$|\text{NewUp}(u)| = |\text{SampleUp}(v)| + |\text{SampleUp}(w)| = \frac{1}{4}(|\text{Up}(v)| + |\text{Up}(w)|) = \frac{1}{2}|\text{Up}(v)|$, which gives $|\text{Up}(u)| = \frac{1}{4}|\text{Up}(v)|$. Also, the number of nodes at u's level is the number of nodes at v's level divided by 2. This implies that the total size of the Up arrays at u's level is $\frac{1}{8}$ of the total size of the Up arrays at v's level, if v is not external. This is also true for the first stage in which nodes at v's level are external. But for the second stage, the nodes at v's level produce samples by picking every second item, which implies $|\text{Up}(u)| = \frac{1}{2}|\text{Up}(v)|$. Similarly, for the third stage we have $|\text{Up}(u)| = |\text{Up}(v)|$. To summarise:

Observation 4.6
The total size of the Up arrays at u's level is $\frac{1}{8}$ of the total size of the Up arrays at v's level, if v is not external, or if v is in its first stage as external. The fraction is $\frac{1}{4}$ if v is in its second stage as external, and it is $\frac{1}{2}$ if v is in its third stage as external.

This gives the following upper bound for the total size of the Up arrays[10]

$$\sum_u |\text{Up}(u)| \le n + n/2 + n/16 + n/128 + \ldots (= 11n/7) \qquad (4.6)$$

The total size of the SampleUp arrays for a given level equals $\frac{1}{4}$ of the total size of the Up arrays at the same level in the normal case. As before, we have two exceptions given by the different sampling rates used when nodes

[10] $\sum_u$ denotes a summation over all active nodes u. The right part of the equation has been put into parentheses to remind the reader that the = sign only is valid when n is infinitely large.

are in their second or third stage as external. The third stage as external gives the largest total size, so we have the following upper bound

$$\sum_u |\text{SampleUp}(u)| \leq n + n/8 + n/64 + n/512 + \ldots (= 8n/7) \qquad (4.7)$$

The NewUp arrays contain the items in the SampleUp arrays. They are located one level higher up in the tree, but the total size becomes the same

$$\sum_u |\text{NewUp}(u)| = \sum_u |\text{SampleUp}(u)| \qquad (4.8)$$

4.3.1.3 A Sliding Pyramid of Processors

Cole describes that one should have one processor standing by each item in the Up, NewUp, and SampleUp arrays. This strategy implies a *processor requirement* which is bounded above by $11n/7 + 2(8n/7) = 27n/7$ which is slightly less than $4n$.

Consider the Up arrays, and the expression given for its size in Equation 4.6. There are n processors (array items) at the lowest active level, a maximum of $n/2$ processors at the next level above, a maximum of $n/16$ processors at the level above that, and so on. This may be viewed as a *pyramid of processors*. Each time the lowest *active* level moves one level up—the pyramid of processors follows so that we still have n processors at the lowest active level. Similarly, the NewUp processors (and SampleUp processors) may be viewed as a pyramid of processors that slides up towards the top of the tree during the progress of the algorithm.

4.3.1.4 Processor Allocation Snapshot

Since the processor allocation is so crucial for the whole implementation, we illustrate it by a simple example. Consider a problem size of $n = 16$. In this case, Equations 4.6 to 4.8 imply a maximum processor requirement of 61 processors. As discussed above, this requirement may only occur in every third stage. Nevertheless, since the main goal for the implementation was to obtain a simple and fast program, not to minimise the processor requirement, we allocate 61 processors once at the start of the program.

We get three "pyramids" of processors as shown in Figure 4.11. The distribution into levels is static. Thus processor no 17 is always the lowest

```
Up processors:
                           25
                17 18 19 20 21 22 23 24
    01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16

SampleUp processors:
                         42 43
    26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41

NewSampleUp processors:
                         60 61
    44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59
```

Figure 4.11: Cole's algorithm, $n = 16$. 25 Up processors, 16 SampleUp processors and 16 NewSampleUp processors distributed into levels.

numbered processor used for the Up arrays one level above the external nodes.

Figure 4.12 shows how these processor pyramids are allocated to the binary tree nodes during stage 8. (The nodes in the binary tree are numbered as depicted in Figure 4.13.) The snapshot illustrates several situations which must be handled.

Not all levels of the processor pyramids are active in every stage, recall Figure 4.10. The set of *active levels* change during the computation. All nodes that are situated at active levels are called *active nodes.* In some stages, the size of the arrays is smaller than its maximum size at that level. Since we have allocated enough processors to cover the maximum size, there may exist levels with some *active* and some *passive processors* (*e.g.* the bottom level of the SampleUp pyramid in Figure 4.12). Since the processor pyramid "slides" towards the top of the binary tree during the computation, the number of nodes covered by a certain level of a processor pyramid will vary. Consequently, the processors must be distributed to the nodes dynamically. As an example, in stage 8 the lowest level of the Up processors is allocated to the four nodes 4–7. In stage 11, the same processors are distributed on the two nodes 2 and 3. Further, since the size of the arrays at a certain level may change between two stages, the allocation of processors to array items within a node must be done at the start of each stage. As an example, processor no 19 (an Up processor) is allocated to item 1 in node 3 in stage 8, but to item 3 in node 2 in stage 9.

```
Up processors:              P
                 2  2  3  3  P  P  P  P
     4  4  4  4  5  5  5  5  6  6  6  6  7  7  7  7

SampleUp processors:      P  P
     4  4  5  5  6  6  7  7  P  P  P  P  P  P  P  P

NewUp processors:         P  P
     2  2  2  2  3  3  3  3  P  P  P  P  P  P  P  P
```

Figure 4.12: Snapshot of processor allocation taken during stage 8 when sorting 16 items. For each processor (see Figure 4.11) it is shown which node the processor is allocated to. Legend: i indicates that the processor is allocated to node i. `P` indicates that the processor is passive.

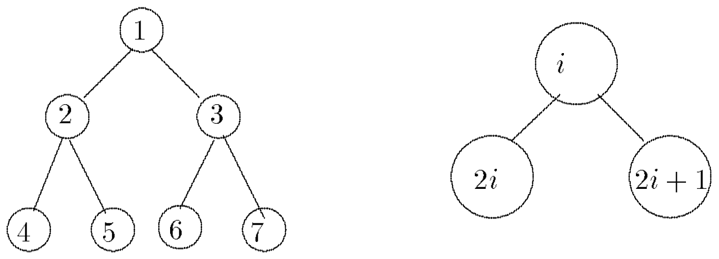


Figure 4.13: "Standard" numbering of nodes in binary tree used throughout this thesis.

```
     CREW PRAM **procedure** *ColeMergeSort*
     **begin**
(1)      Compute the processor requirement, *NoOfProcs*;
(2)      Allocate working areas;
(3)      Push addresses of working areas and other facts on the stack;
(4)      **assign** *NoOfProcs* processors;
(5)      **for each processor do begin**
(6)          Read facts from the stack;
(7)          *InitiateProcessors*;
(8)          **for** $Stage := 1$ **to** $3 \log n$ **do begin**
(9)              *ComputeWhoIsWho*;
(10)             **if** $Stage > 1$ **then** *CopyNewUpToUp*;
(11)             *MakeSamples*;
(12)             *MergeWithHelp*;
             **end**;
         **end**;
     **end**;
```

Figure 4.14: Main program of Cole's parallel merge sort expressed in Parallel Pseudo Pascal (PPP).

4.3.2 Pseudo Code and Time Consumption

Cole's succinct description of the algorithm is at a sufficiently high level to give the programmer freedom to choose between the SIMD or MIMD [Fly66] programming paradigms. The algorithm has been programmed in a *synchronous MIMD programming style*, as proposed for the PRAM model by Wyllie [Wyl79]. It is written in PIL and the program is executed on the CREW PRAM simulator.

We will use the PPP notation (Section 3.2.2) to express the most important parts of the implementation in a compact and readable manner. (A complete PIL source code listing is provided in Appendix B.)

4.3.2.1 The Main Program

The main program of Cole's parallel merge sort algorithm is shown in Figure 4.14. When the program starts, one single CREW PRAM processor is running, and the problem instance and its size, n, are stored in the global

Table 4.5: Time consumption for the statements in the main program of *Cole-MergeSort* shown in Figure 4.14. (The entries for statements (9–12) are valid only for stage 7 and later stages. The time used is shorter for some of the early stages. See Section 4.3.2.4 at page 148.)

$t(1,n)$	$=$	$34 + 8\lfloor \log_8(n/2) \rfloor + 8\lfloor \log_8 n \rfloor$
$t(2..3,n)$	$=$	82
$t(4,n)$	$=$	$42 + 23\lfloor \log \mathit{NoOfProcs} \rfloor$
$t(5..6,n)$	$=$	13
$t(7,n)$	$=$	$224 + 36\lfloor \log_8(n/2) \rfloor + 72\lfloor \log_8 n \rfloor$
$t(8..9,n)$	$=$	94
$t(10,n)$	$=$	52
$t(11,n)$	$=$	47
$t(12,n)$	$=$	776

memory. Statements (1–3) are executed by this single processor alone.

Computing the processor requirement

The maximum processor requirement, here denoted *NoOfProcs*, is given by the sum of the maximum size of the Up(u), NewUp(u) and SampleUp(u) arrays for all nodes u during the computation. Thus we have:

$$\mathit{NoOfProcs} = \sum_u |\mathrm{Up}(u)| + \sum_u |\mathrm{NewUp}(u)| + \sum_u |\mathrm{SampleUp}(u)| \qquad (4.9)$$

For a given finite n, the *exact* calculation of *NoOfProcs* is done by replacing the $\leq$ sign with $=$ in Equation 4.6 and 4.7, and summing terms as long as n is not smaller than the divisor. A simple way to implement these calculations is to use "divide by 8 loops".

The resulting time consumption for statement (1) as a function of the problem size is given in Table 4.5 as $t(1,n)$.

Working areas and address broadcasting

When the total number of processors has been computed, it is possible to allocate working areas of the appropriate size in the PRAM global memory. These are dedicated areas which are used to exchange various kinds of information between the processors, see Figure 4.15.

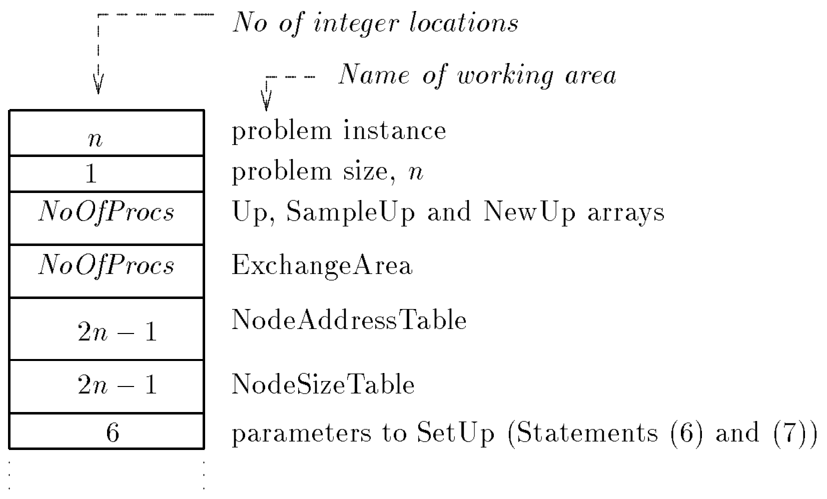


Figure 4.15: Memory map for implementation of Cole's algorithm. *NoOfProcs* locations are used to store the contents of the Up, SampleUp and NewUp arrays—one integer location for each array item (processor). The ExchangeArea is used to communicate array items between the processors. The NodeAddressTable and NodeSizeTable are used to help the processors find the address and size of the arrays of other nodes in the tree.

Statement (3) describes how central facts such as the addresses of the working areas are broadcasted from the single master processor. It is done by pushing values on the (user) stack in the PRAM global memory. When all *NoOfProcs* processors have been allocated and activated, they may read these data in parallel from the stack (Statement (6)). Due to the *concurrent read* property of the CREW PRAM this kind of broadcasting is easily performed in $O(1)$ time.

Processor allocation and activation

The processors are allocated with the standard procedure described in CREW PRAM property 7 at page 67. Since *NoOfProcs* is $O(n)$ the expression given for $t(4,n)$ in Table 4.5 is $O(\log n)$.

The desire for developing a simple and fast implementation did also influence the structure of processor allocation and activation.

Implementing "sliding processors"

Recall the description given in Section 4.3.1 of how the processors should be allocated to the binary computation tree. When implementing this, it is crucial to split the necessary computation into a *static* and *dynamic* part. The dynamic part, called *ComputeWhoIsWho* (see page 137), is executed at the beginning of each stage. Consequently, it must be done in $O(1)$ time—to avoid spoiling the $O(\log n)$ time overall algorithm performance. This is possible since the most "difficult" part need only be computed once initially. The static part is represented by statement (7).

InitiateProcessors, which is executed by all processors in parallel, starts by computing the address in the global memory of the (Up, SampleUp or NewUp) array item associated with the processor. It reads the problem size n, computes the number of stages, and decides its own processor type (Up, SampleUp or NewUp). Then, the Up processors read one input number each and place it in its own Up array item.

Then, it is time to perform the static part of the "processor sliding". It is done level by level for each processor type. It is calculated which level in the "pyramid" (See Figure 4.11) the processor is assigned to. This is counted in number of levels (zero or more) above the lowest active level (which always contains the external nodes). Thus, whether a processor is assigned to an inside node or an external node is also known at this point. Finally, the "local" processor number *at that level*, numbered from the left, is computed. Again, the level by level approach is implemented by "divide by 8 loops" giving the logarithmic terms for $t(7, n)$ shown in Table 4.5.

4.3.2.2 The Main Loop—Implementation Details

The $3 \log n$ stages each consists of four main computation steps.

ComputeWhoIsWho

ComputeWhoIsWho starts by calculating the lowest and highest active level in the binary computation tree (recall Figure 4.10). Together with the "level offset" computed initially (*InitiateProcessors*) this gives the exact level for each processor in the current stage. Knowing the stage and the level, it is possible for each (Up, SampleUp and NewUp) processor to compute the size of the (Up, SampleUp or NewUp) array in each node at that level. This again makes it possible for each processor to calculate the node number, and item number within the array for that node, and whether the processor is active or passive during the stage. All the calculations may be done in $O(1)$ time.

CopyNewUpToUp

CopyNewUpToUp transfers the array contents and ranks computed for the NewUp arrays in the previous stage to the corresponding Up arrays in the current stage. The procedure is quite simple, but it will be described in some detail to illustrate how data typically are communicated between the processors.

The procedure is outlined in Figure 4.16. It assumes that the NodeAddressTable (Figure 4.15) for each node in the tree stores the address of the first item in the NewUp array for that node during the previous stage.[11]

The local variable *CorrNewUpItem* is used by each Up processor to hold the address of the NewUp processor[12] in the previous stage which *corresponds* to the Up item associated with the Up processor in the current stage. The processor activation in statement (1) is realised by selecting all active Up processors which are assigned to an inside node. In every third stage,

[11]This assumption is made valid by letting each processor which is assigned to the first item of a NewUp array store the address of that item in the NodeAddressTable at the end of each stage.

[12]*I.e.*, the address in the dedicated part of the global memory used to store the Up, SampleUp, and NewUp arrays, see Figure 4.15.

```
    CREW PRAM **procedure** *CopyNewUpToUp*
    **begin**
        **address** *CorrNewUpItem*;
(1)     **for each processor** assigned to an Up array element **where** the
            NewUp array for the same node was updated in the
            previous stage **do begin**
(2)             *CorrNewUpItem* :=
                    NodeAddressTable[this node]+ this item no −1;
(3)             Up[this processor] := NewUp[*CorrNewUpItem*];
        **end**;
(4)     **for each processor** of type NewUp **do**
(5)         ExchangeArea[this processor] := *RankInLeftSUP*;
(6)     **for each processor** assigned to an Up array element **where** the
            NewUp array for the same node was updated in the
            previous stage **do**
(7)             *RankInLeftSUP* := ExchangeArea[*CorrNewUpItem*];
        { Transfer *RankInRightSUP* in the same manner }
    **end**;
```

Figure 4.16: *CopyNewUpToUp* outlined in PPP.

when the external nodes are in their first stage *as external*, the Up processors assigned to the external nodes should also be selected—since these nodes were inside nodes in the *previous* stage.

In statement (2), the Up processors compute *CorrNewUpItem* in parallel. The *concurrent read* property of the CREW PRAM is exploited since several Up processors are assigned to the same node. The copying of all NewUp arrays to the Up arrays is then done simultaneously by the Up processors (statement (3)). The possibility of doing many writes simultaneously to different locations of the global memory is used. No write conflict will occur because every Up processor is assigned to one unique Up array element.

Knowing the mapping from an Up processor in the current stage to the corresponding NewUp processor in the previous stage, it is straightforward to communicate the ranks computed at the end of each stage to keep Assumption 4.1 at page 118 valid. In the description of Phase 2 – Step 2, maintaining ranks at page 125, it was shown how to compute NewUp(u) $\rightarrow$ NewSampleUp(v) and NewUp(u) $\rightarrow$ NewSampleUp(w). For each NewUp and Up processor these two ranks are stored in the local variables *RankInLeftSUP* and *RankInRightSUP* respectively.[13] Thus Up(u) $\rightarrow$ SampleUp(v) and Up(u) $\rightarrow$ SampleUp(w) will be known in the current stage if we copy these two variables from each NewUp processor to the corresponding Up processor.

The <u>ExchangeArea</u> (Figure 4.15) is allocated for this kind of communication. First, each NewUp processor writes the value of *RankInLeftSUP* to the unique location associated to that processor (statement (5)). Then, since we already know the mapping, the Up processors simply read the new value of *RankInLeftSUP* from the appropriate position in the <u>ExchangeArea</u> (statement (7)). *RankInRightSUP* is transferred in the same way.

CopyNewUpToUp is performed in constant time on a CREW PRAM.

MakeSamples

MakeSamples is a procedure performed by the Up and SampleUp processors in cooperation to produce the samples SampleUp(x) for all active nodes x. This is a straightforward task (see Section 4.2.2.2), and can easily be done in constant time.

First, the Up processors write the address of the first item of the Up array in each active node to the <u>NodeAddressTable</u>. Then, every active

[13] SUP is short for SampleUp (and NUP for NewUp).

```
    CREW PRAM **procedure** *DoMerge*
    **begin**
(1)     **for each processor** of type Up or SampleUp **do**
(2)         *ComputeCrossRanks*;
(3)     **for each processor** of type SampleUp **do begin**
(4)         $RankInParentNUP := RankInThisSUP + RankInOtherSUP$;
(5)         Compute *SLrankInParentNUP* and *SRrankInParentNUP*;
(6)         Find address of the NewUp array in the parent node
                and store it in *ParentNUPaddr*;
(7)         $WriteAddress := ParentNUPaddr + RankInParentNUP - 1$;
(8)         NewUp[*WriteAddress*] := own value;
(9)         Record the sender address for each new NewUp item
                in the ExchangeArea;
        **end**;
    **end**;
```

Figure 4.17: *DoMerge* outlined in PPP.

SampleUp processor calculates the *SampleRate* (4, 2 or 1) to be used to produce the samples *at that level*. Every SampleUp processor knows the node x and item number (index) i within SampleUp(x) it is assigned to. It reads the address of the first item in Up(x) from the NodeAddressTable, and uses the *SampleRate* to find the address of the item in Up(x) which corresponds to item i in SampleUp(x). Finally and in parallel, each active SampleUp processor copies this Up array item to "its own" SampleUp array item.

4.3.2.3 Merging in Constant Time—Implementation Details

Having read the entire algorithm description in Section 4.2 it should be no surprise that the implementation of Phase 2, merging in constant time, constitutes the major part of the implementation. *MergeWithHelp* is split into two parts. *DoMerge* which performs Step 1 of the algorithm (recall Figure 4.6 at page 120), and *MaintainRanks* which performs Step 2.

DoMerge

DoMerge is outlined in Figure 4.17. Its main task is to compute the position of each SampleUp array item in the NewUp array of its parent node, (See Figure 4.5 at page 119). This rank is stored in the variable *RankInParentNUP*. It is given as the sum of the rank of the item in the SampleUp array which itself is a part of, and its rank in the SampleUp array of its sibling node. These two ranks are stored in the variables *RankInThisSUP* and *RankInOtherSUP* respectively, and they are computed by the procedure *ComputeCrossRanks*.

Postpone statement (5) for a moment. After having computed *RankInParentNUP* the SampleUp processors must get the address of the NewUp array of the parent node, as expressed by statement (6). It is done by using the NodeAddressTable in the same way as in the procedure *CopyNewUpToUp* described above. Then, each SampleUp processor knows the right address in the NewUp array for writing its own item value (statement (8)). Notice that all the items in all the NewUp arrays are updated simultaneously. We do not get any write conflicts, but surely the simultaneous writing of n values performed by n different processors represents an intensive utilisation of the CREW PRAM global memory.

Before *DoMerge* terminates, the SampleUp processors must also store the sender address for each NewUp item just written. This information is needed in *MaintainRanks*. The sender address of an item is represented by the item no (index) in the SampleUp array it was copied from, and it is stored in the ExchangeArea. If the SampleUp item is situated in a left child node, the address is stored as a negative number.

Now, let us return to statement (5). Repeat Assumption 4.2 at page 126 which states that for each SampleUp array item we should know the straddling items from the other SampleUp array. In the terms used in Figure 4.9—for each item e we must know d and f. These positions are possible to compute now, because *RankInParentNUP* has been computed by all the SampleUp processors.

The calculation is based on the two variables *SLindex* and *SRindex*[14] computed by *ComputeCrossRanks*. For an item e in SampleUp(v) (see Figure 4.8) *SLindex* and *SRindex* are the positions in SampleUp(w) of the two items in SampleUp(w) that would straddle e if e was inserted according to sorted order in SampleUp(w). Thus in general we have *SLindex* = *RankInOtherSUP* and *SRindex* = *RankInOtherSUP* + 1.

[14]SL is short for straddling on the left side, and SR for straddling on the right side.

Again we use the ExchangeArea. All SampleUp processors write their own value of *RankInParentNUP* to it. Then, *SLindex* is used to find the address in the ExchangeArea of the SampleUp item from the sibling node that would straddle this item on the left side. The position of this item in NewUp of the parent node, called *SLrankInParentNode* is then read from the ExchangeArea. Similarly, *SRindex* gives the address for reading *SRrankInParentNode*—and we have reached our goal which was to know the positions of d and f for each item e in NewUp(u) in Figure 4.9. Note that *SLrankInParentNUP* and *SRrankInParentNUP* are stored locally in each processor and need not be communicated to other processors before they are used in *MaintainRanks*.

ComputeCrossRanks

ComputeCrossRanks is outlined in Figure 4.18. It is performed by the Up and SampleUp processors in cooperation. As explained in Section 4.2.2.4 *RankInThisSUP*[15] is easily found (statement(2)). However, the calculation of *RankInOtherSUP* is rather complicated and split into two substeps.[16]

The pseudocode for Substep 1 describes that all left child nodes (v) are handled first, and then all right child nodes (w). In other cases (such as statements (4–9) in *DoMerge*) both SampleUp arrays are handled in parallel. The reason for the reduced parallelism in this case is that the work represented by statements (4) and (5) is done by the Up processors in their *common* parent node.[17]

Recall Figure 4.7 at page 121. A description of the code that implements statement (4) follows. The Up processors starts by writing its value of Up(u) $\rightarrow$ SampleUp(v) to the ExchangeArea. Then, in addition to knowing r, the Up processors may read the value s from the ExchangeArea. Further, in addition to knowing its own value i_1, it reads the value of i_2 stored in the Up array. The address of the SampleUp(v) array is found by using the NodeAddressTable. Now, the situation is exactly as in Figure 4.7, and each

[15]The variable *RankInThisSUP* stores R(e,SampleUp(x)) where $e \in$ SampleUp(x) and $x = v$ or $x = w$. The rank of an item e in the SampleUp array of its sibling node is stored in *RankInOtherSUP*.

[16]An exception which is much simpler than the general case occurs in nodes with SampleUp arrays containing only one item. This implies that there is no Up array in the parent node to help us in computing the rank in the SampleUp array of the sibling node. However, since that array also contains only one item, *RankInOtherSUP* may be computed by doing one single comparison.

[17]Can you suggest a possible strategy for handling both child nodes in parallel?

```
     CREW PRAM procedure ComputeCrossRanks
     begin
(1)      for each processor of type SampleUp do
(2)          RankInThisSUP := position (index) of item in own array;
(3)      for each processor of type Up do begin
             { Substep 1: (Section 4.2.2.6, page 121) }
(4)          For each item $e$ in SampleUp($v$) compute its rank in Up($u$);
(5)          For each item $e$ in SampleUp($w$) compute its rank in Up($u$);
             {RankInParentUP is now stored in the ExchangeArea }
         end;
(6)      for each processor of type Up or SampleUp do begin
             { Substep 2: (Section 4.2.2.7, page 122) }
(7)          if processor type is SampleUp then
(8)              Read RankInParentUP from the ExchangeArea;
(9)          if processor type is Up then begin
(10)             Store address/size into the NodeAddress/SizeTable;
(11)             Store RankInRightSUP into the ExchangeArea;
             end;
(12)         if processor type is SampleUp in a left child node then
(13)             Find $r$ and $t$;
(14)         if processor type is Up then
(15)             Store RankInLeftSUP into the ExchangeArea;
(16)         if processor type is SampleUp in a right child node then
(17)             Find $r$ and $t$;
(18)         if processor type is SampleUp then begin
(19)             Read maximum 3 values from the other SampleUp array;
(20)             LocalOffset := position of own value among these 3;
(21)             RankInOtherSUP := $r$ + LocalOffset;
             end;
         end;
     end;
```

Figure 4.18: *ComputeCrossRanks* outlined in PPP.

Up processor considers the candidates in positions r to s according to the rules described in Section 4.2.2.6. When the Up processor which represents item i_1 (Figure 4.7) has found that an item in SampleUp(v) is contained in the interval induced by i_1, denoted $I(i_1)$, it assigns the index of i_1 to the variable *RankInParentUP* of that SampleUp item. This assignment is done indirectly by writing the value to the ExchangeArea. At the start of Substep2, the SampleUp processors read this value.

The SampleUp item pointed to by s for i_1 is the same as the item pointed to by r for i_2. Consequently, these "border items" are considered by two processors. Therefore, one might think that our strategy for updating *Rank-InParentUP* may lead to write conflicts. However, this will not occur since the Up processor associated with an item x only assigns to the value *Rank-InParentUP* for those items in SampleUp(v) which are contained in $I(x)$, and every SampleUp item is member of exactly one such set $I(x)$.

Substep 2 is performed by the SampleUp and Up processors in cooperation (statement (6)). Again, we handle all left child nodes before all right child nodes (statements (12) and (16)). The left child nodes must access the (local Up) variable *RankInRightSUP* to find the values of r and t, and the right child nodes must access *RankInLeftSUP*. The SampleUp processors use the NodeAddressTable and the variable *RankInParentUP* to locate the items d and f (Figure 4.8 page 123). The values of r and t is then read from the ExchangeArea at the positions corresponding to d and f (statements (13) and (17)).

When all SampleUp processors have calculated r and t they may do the rest of Substep2 in parallel (statement (18)). They read the values of the items in positions $[r+1, t]$ in the SampleUp array of their sibling node, and compare their own value with these at most three values. The position with respect to sorted order is found and stored in *LocalOffset*. *RankInOtherSUP* is then given as shown in statement (21).

MaintainRanks

As described in Section 4.2.2.8 the computation of NewUp(u) $\rightarrow$ NewSample-Up(v) and NewUp(u) $\rightarrow$ NewSampleUp(w) is split into two cases—here represented by the procedures *RankInNewSUPfromSenderNode* and *Rank-InNewSUPfromOtherNode*. Figure 4.19 shows the most important parts of the "from sender node" case. Both procedures are executed by the Up, SampleUp and NewUp processors in cooperation.

Recall Section 4.2.2.9. The first part of the procedure is performed by the

```
    CREW PRAM procedure RankInNewSUPfromSenderNode
    begin
        {Performed in parallel by Up, SampleUp and NewUp processors}
(1)     if processor type is Up then begin
(2)         RankInThisNUP := RankInLeftSUP + RankInRightSUP;
(3)         ExchangeArea[this processor] := RankInThisNUP;
        end;
(4)     if processor type is SampleUp then begin
(5)         By help of the sample rate used in this stage at this level
                find the position in the Up array corresponding to this
                SampleUp item;
(6)         Use this position to read the correct value of RankInThisNUP
                from the ExchangeArea;
            {RankInThisNUP stores SampleUp($x$) $\rightarrow$ NewUp($x$)}

(7)         Compute RankInThisNewSUP from RankInThisNUP by
                considering the sample rate that will be used in the next
                stage to make NewSampleUp from NewUp;
            {RankInThisNewSUP stores SampleUp($x$) $\rightarrow$ NewSampleUp($x$)}
(8)         ExchangeArea[this processor] := RankInThisNewSUP;
        end;
(9)     if processor type is NewUp then
(10)        if this item came from SampleUp($v$) then
(11)            Read the value of RankInLeftSUP (valid for the next
                    stage) from the position in the ExchangeArea
                    corresponding to its sender item in SampleUp($v$)
(12)        else {item came from SampleUp($w$)}
(13)            Read the value of RankInRightSUP (valid for the next
                    stage) from the position in the ExchangeArea
                    corresponding to its sender item in SampleUp($w$);
        end;
    end;
```

Figure 4.19: *RankInNewSUPfromSenderNode* outlined in PPP. Here we have assumed that the processor activation has been done at the caller place, and not inside the procedure.

Up processors. Since NewUp(u) contains exactly the items in SampleUp(v) and SampleUp(w), Up(u) $\rightarrow$ NewUp(u) is easily computed as expressed in statement (2). The ExchangeArea is used to transfer these values to the SampleUp processors.

In the second part of the procedure the SampleUp processors compute SampleUp(x) $\rightarrow$ NewSampleUp(x), $x \in v, w$. Note that both the left and right child nodes are handled in parallel. Each SampleUp item (processor) calculates its corresponding Up item in the *same* node, so that SampleUp(x) $\rightarrow$ NewUp(x) is known after statement (6) has been executed. Since NewSampleUp(x) is made from NewUp(x), we get SampleUp(x) $\rightarrow$ NewSampleUp(x) which is stored in the local variable *RankInThisNewSUP*, and written to the ExchangeArea.

The last part of *RankInNewSUPfromSenderNode* is performed by the NewUp processors. Remember that we used the ExchangeArea to store the sender address of each NewUp item at the end of procedure *DoMerge* (statement (9) in Figure 4.17). This information is read by the NewUp processors at the very start of *MaintainRanks*, before *RankInNewSUPfrom-SenderNode* is called. We describe statement (11), statement (12) contains different but quite symmetric code.[18] Each NewUp item (processor) uses its sender address to find its corresponding in SampleUp(v). Since the ExchangeArea stores SampleUp(x) $\rightarrow$ NewSampleUp(x) for all nodes x, an arbitrary item e in NewUp(u) that came from SampleUp(v) may indirectly read R(e,NewSampleUp(v)) from the ExchangeArea—so that NewUp(u) $\rightarrow$ SampleUp(v) becomes known for all items in v that came from v. (This rank is stored in the variable *RankInLeftSUP*, and is copied from the NewUp to the Up processors at the beginning of next stage by *CopyNewUpToUp* (Figure 4.16).)

Procedure *RankInNewSUPfromOtherNode* is outlined in Figure 4.20. Almost all the work is done by the NewUp processors. Recall Section 4.2.2.10 and Figure 4.9 at page 127.

The procedure starts by transferring the variables *SL-* and *SRrankInParentNUP* (which were computed in *DoMerge*) from the SampleUp processors to the NewUp processors. Statement (4) describes that the values r and t are communicated from the processors holding item d and f to the processor holding item e. Again the ExchangeArea is used, and we get a reduced

[18]Statements (9–13) give a typical example on how the MIMD property of the CREW PRAM model may be used to reduce the execution time and increase the processor utilisation.

```
    CREW PRAM procedure RankInNewSUPfromOtherNode
    begin
        {Performed in parallel by Up, SampleUp and NewUp processors}
(1)     Communicate SLrankInParentNUP from the SampleUp processors
            to the NewUp processors through the ExchangeArea;
(2)     Communicate SRrankInParentNUP from the SampleUp processors
            to the NewUp processors through the ExchangeArea;
        { Each NewUp processor knows the positions of $d$ and $f$ }

(3)     Use the NodeAddress/SizeTable to let each NewUp processor know
            the address/size of the NewUp array in the other child node;
(4)     For each NewUp item $e$ that came from SampleUp($w$) get the
            values of RankInLeftSUP from the NewUp items $d$ and $f$ that
            came from SampleUp($v$);
(5)     For each NewUp item $e$ that came from SampleUp($v$) get the
            values of RankInRightSUP from the NewUp items $d$ and $f$ that
            came from SampleUp($w$);
        { Each NewUp processor knows the value of $r$ and $t$ }

(6)     Compute the local offset of each NewUp item $e$ within the
            (maximum 3) items in positions $[r+1, t]$ in the NewSampleUp
            array of its other child node; { See the text }
(7)     The rank of NewUp item $e$ in the NewSampleUp array of the other
            node is now given by $r$ and the local offset just computed,
            and it is stored in RankInLeftSUP or RankInRightSUP;
    end;
```

Figure 4.20: *RankInNewSUPfromOtherNode* outlined in PPP.

degree of parallelism as implicitly expressed by statements (4) and (5). It is statement (4) that corresponds to Figure 4.9. Further note that the ranks r and t correspond to the "from sender node case" which just has been solved.

For each item e in NewUp(u), we must locate the items in the positions $[r+1, t]$ in NewSampleUp(v) (or NewSampleUp(w)). This is *not* trivial because the NewSampleUp arrays do not exist. However, we know that NewSampleUp(x) will be made by the procedure *MakeSamples* from NewUp(x) in the next stage. Thus, a given item α in NewSampleUp(v) (or NewSampleUp(w)) may be located in NewUp(v) (or NewUp(w)) if we take into account the sampling rate that will be used at that level (one level below) in the next stage.

This should explain how it is possible to implement statement (6), and why we bothered to store the addresses of the NewUp arrays in statement (3).

Finally, each NewUp item (processor) that came from SampleUp(w) (SampleUp(v)) computes its rank in NewSampleUp(v) (NewSampleUp(w)), and stores this value in *RankInLeftSUP* (*RankInRightSUP*).

4.3.2.4 Performance

The time used to perform a *Stage* (statements (9)–(12) in Figure 4.14 at page 133) is somewhat shorter for the six first stages than the numbers listed in Table 4.5 at page 134. This is because some parts of the algorithm do not need to be performed when the sequences are very short. However, for all stages after the six'th, the time consumed is as given by the constants in the table. Stages 1–6 takes a total of 2452 time units. The total time used by *ColeMergeSort* on n distinct items, $n = 2^m$, m is an integer, and $m > 1$, may be expressed as

$$T(\textit{ColeMergeSort}, n) = t(1..7, n) + 2452 + t(8..12, n) \times 3((\log n) - 2) \quad (4.10)$$

The $O(1)$ time merging performed by *MergeWithHelp* constitutes the major time consumption of the algorithm. Of the time used by *MergeWithHelp* (776 time units), about 40% is needed to compute the *crossranks* (Substep 1 and 2, p. 773, [Col88]), and nearly 43% is used to *maintain ranks* (Step 2, p. 774). Note that the execution time of Cole's parallel merge sort algorithm is independent of the actual problem instance for a given problem size. Equation 4.10 makes it possible to calculate the time consumption of the algorithm as an exact figure for any problem size (which is a power of 2).

Table 4.6: Performance data measured from execution of *ColeMergeSort* on the CREW PRAM simulator. Time and cost are given in *kilo* CREW PRAM time units and *kilo* CREW PRAM (unit-time) instructions respectively. The number of read/write operations to/from the global memory are given in *kilo* locations.

Problem size n	time	#processors	cost	#reads	#writes
4	2.9	14	40.8	0.3	0.2
8	5.9	30	177.8	1.0	0.6
16	8.9	61	542.9	2.8	1.8
32	11.8	122	1443.3	7.4	4.8
64	14.8	246	3650.6	18.5	11.9
128	17.8	493	8795.6	44.7	28.2
256	20.7	986	20453.6	104.6	65.2

Table 4.6 shows the performance of Cole's algorithm on various problem sizes.

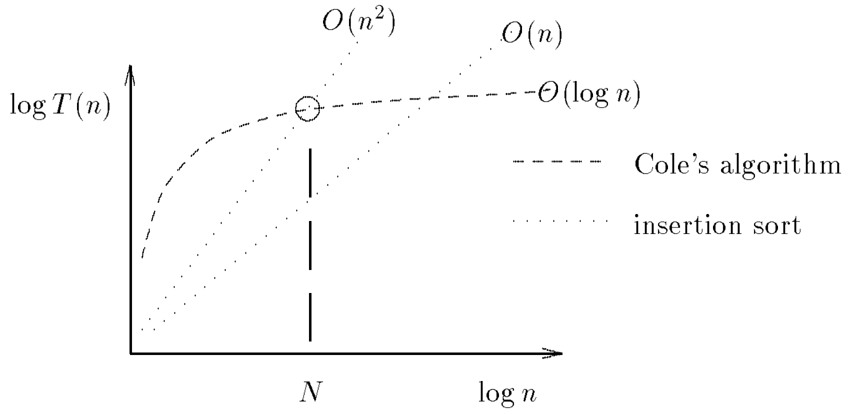


Figure 4.21: Outline of results expected when comparing Cole's parallel merge sort with straight insertion sort.

4.4 Cole's Parallel Merge Sort Algorithm Compared with Simpler Sorting Algorithms

4.4.1 The First Comparison

When starting on this work, I expected that Cole's algorithm was so complicated that it in spite of its known $O(\log n)$ time complexity would be slower than simpler parallel or sequential algorithms—unless we assumed *very large* problem sizes. I "planned" to show that even straight insertion sort executed on one single processor would perform better than Cole's algorithm as long as $n < N$, and I expected N to be so large that it would imply a processor requirement for Cole's algorithm in the range of hundreds of thousands, or millions of processors. *Then*, I could have claimed that Cole's algorithm is of little practical value.

The kind of results that I expected is illustrated in Figure 4.21. The point where Cole's algorithm becomes faster than 1-processor insertion sort for worst case problems[19] is marked with a circle in the figure. The corresponding problem size is denoted N. Since insertion sort is *very* simple, I expected that the relatively high descriptional complexity of Cole's algorithm would imply N to be as large as 10^5, or even larger.

However, the results for the time consumption of Cole's algorithm were

[19] *I.e.* problems making insertion sort to an $O(n^2)$ algorithm.

much better than expected. Figure 4.22 shows the time used to sort n integers by Cole's algorithm compared with 1-processor insertion sort and n-processor odd-even transposition sort. Note that we have logarithmic scale on both axes.

Cole's algorithm is the CREW PRAM algorithm described in the previous section, and with performance data as summarised in Table 4.6. *Insertion sort* is the algorithm which was described in Section 3.3.2.2. Its performance was described in Section 4.1.2.1. The simplest version of *odd-even transposition sort* algorithm was described in Section 3.3.2.1. Here we are studying the version developed in Section 3.4 which requires only $n/2$ processors to sort n items. The performance of that CREW PRAM implementation was reported in Section 4.1.2.3. The time used by Cole's algorithm and odd-even transposition sort are independent of the actual problem instance for a fixed problem size. However, insertion sort requires $O(n^2)$ time in the worst case, and $O(n)$ time in the best case (both shown in the figure).

We see that Cole's algorithm becomes faster than insertion sort (worst case) for $N \approx 64$. Also, Cole's algorithm becomes faster than odd-even transposition sort (which is very simple—and uses $O(n)$ processors) for problem sizes at about 4096 items.

These somewhat surprising results (Figure 4.22) were presented at the Norwegian Informatics Conference i 1989 [Nat89]. It was a preliminary comparison which did *not* support my expectations. The practical value of Cole's algorithm was still an open question, and it was clear that further investigations were needed.

In retrospect, being surprised by the obtained results can be considered as an underestimation of the difference between polynomials such as n^2 and n, and $\log n$ for large and even modest values of n.

4.4.2 Revised Comparison Including Bitonic Sorting

A natural candidate for further investigations was Batcher's bitonic sorting described in Section 4.1.3. Its relative simplicity together with a $O(\log^2 n)$ time complexity may explain claims such as "*nobody beats bitonic sorting*" [San89b].

In addition to implementing bitonic sorting, all the compared algorithm implementations were revised and some improvements were done. The modelling of the time consumption in the various implementations was also revised in order to give a fair comparison. The final results, shown in Figure

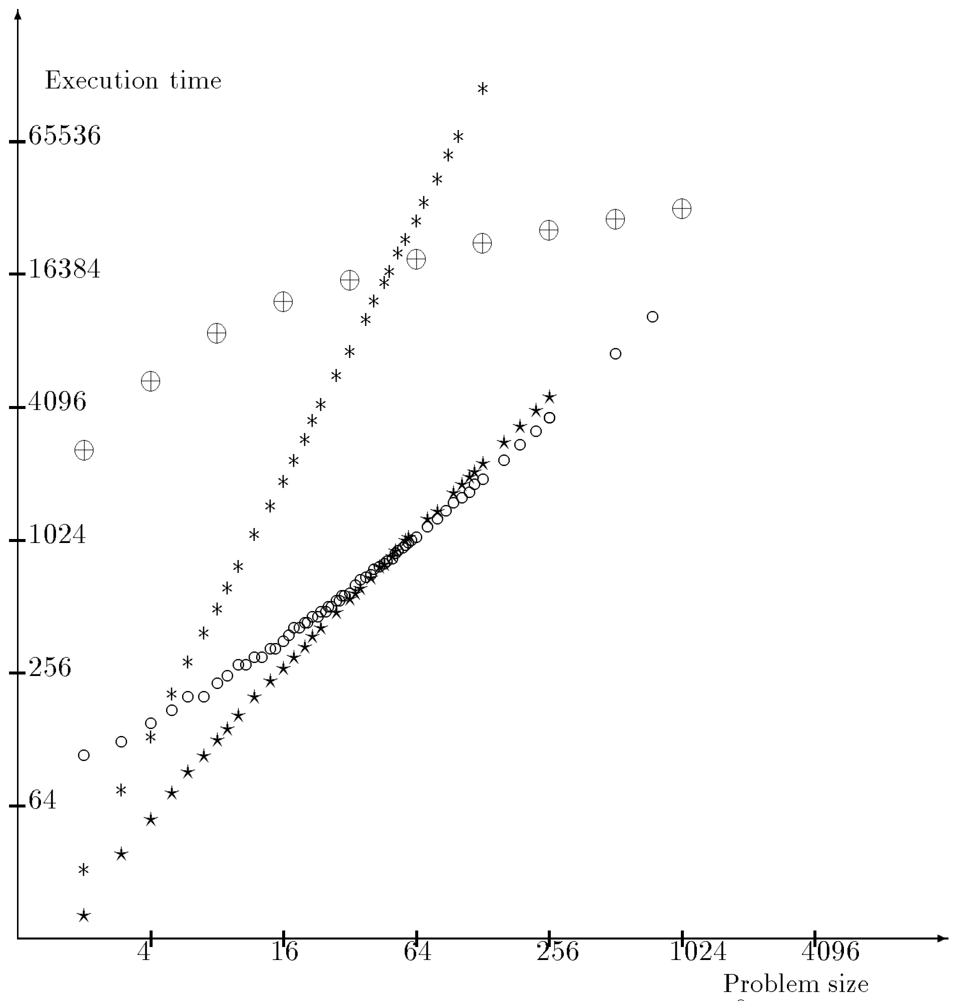


Figure 4.22: Comparison of Cole's algorithm with $O(n)$ and $O(n^2)$ time algorithms [Nat89]. Legend: $\oplus$ = cole's algorithm ($O(\log n)$), $\circ$ = odd-even transposition sort ($O(n)$), $*$ = insertion sort (worst case, $O(n^2)$), and $\star$ = insertion sort (best case, $O(n)$).

Table 4.7: Performance data for the CREW PRAM implementations of the studied sorting algorithms. Problem size n is 128. Time and cost are given in *kilo* CREW PRAM time units and *kilo* CREW PRAM (unit-time) instructions respectively. The number of read/write operations to/from the global memory are given in *kilo* locations.

Algorithm	time	# processors	cost	# reads	# writes
Cole	17.8	493	8795.6	44.7	28.2
Bitonic	2.6	64	169.0	3.9	3.7
Odd-Even	2.8	64	176.5	16.6	8.0
Insert-worst	148.7	1	148.7	8.3	8.3
Insert-Average	76.2	1	76.2	4.3	4.2
Insert-best	2.8	1	2.8	0.3	0.1

Table 4.8: Same table as above but with problem size $n = 256$.

Algorithm	time	# processors	cost	# reads	# writes
Cole	20.7	986	20453.6	104.6	65.2
Bitonic	3.4	128	428.2	9.9	9.4
Odd-Even	5.3	128	683.7	65.9	34.8
Insert-worst	592.4	1	592.4	32.9	32.9
Insert-Average	303.3	1	303.3	17.0	16.8
Insert-best	5.6	1	5.6	5.1	0.3

4.23, were presented at the SUPERCOMPUTING'90 conference [Nat90b]. We see that bitonic sorting is fastest in this comparison in a large range of n starting at about 256.

Table 4.7 and 4.8 show some central performance data for small test runs, $n = 128$ and $n = 256$. The two rightmost columns show the total number of read operations from the global memory and the total number of write operations to the global memory.

The implementation of *ColeMergeSort* counts about 2000 PIL lines and was developed and tested in about 40 days of work, see Appendix B. The corresponding numbers for *BitonicSort* are about 200 PIL lines and roughly 2 days of work. The listing is given in Appendix C.

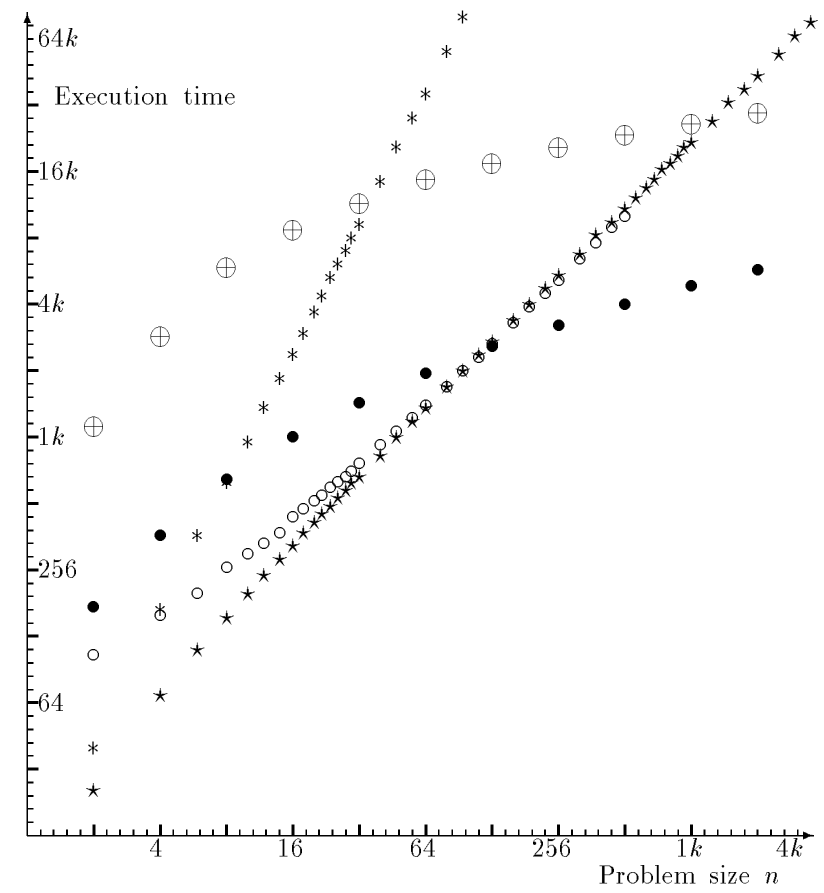


Figure 4.23: Time consumption (in number of CREW PRAM time units) measured by running parallel sorting algorithms on the CREW PRAM simulator for various problem sizes n (horizontal axis). Note the logarithmic scale on both axes. Legend: $\oplus$ = Cole's algorithm ($O(\log n)$), $\bullet$ bitonic sorting ($O(\log^2 n)$), $\circ$ = odd-even transposition sort ($O(n)$), $*$ = insertion sort (worst case, $O(n^2)$), and $\star$ = insertion sort (best case, $O(n)$).

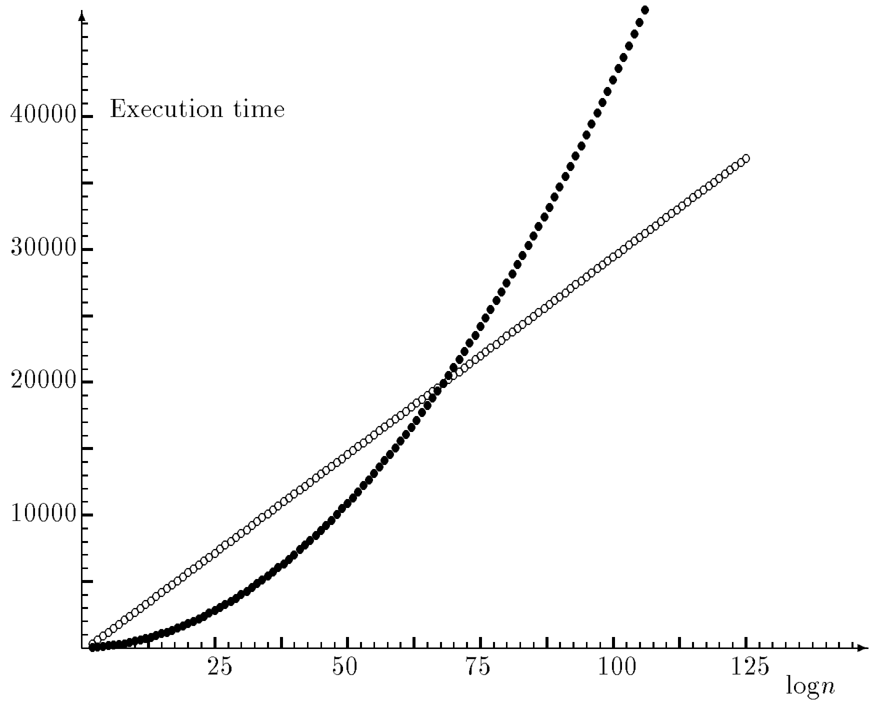


Figure 4.24: Comparison of time consumption in the CREW PRAM implementations of Cole's parallel merge sort algorithm (marked with ∘) and Batcher's bitonic sorting algorithm (marked with •).

4.4.2.1 Bitonic Sorting Is Faster In Practice!

As shown in Section 4.1.3.1 and 4.3.2.4 exact analytical models have been developed for the time consumption in the implementations of Cole's algorithm and bitonic sorting. The models have been checked against the test runs, and have been used to find the point where the CREW PRAM implementation of Cole's $O(\log n)$ time algorithm becomes faster than the CREW PRAM implementation of $O(\log^2 n)$ time bitonic algorithm. See Figure 4.24.

The results are summarised in Table 4.9. The table shows time and processor requirements for the two algorithms for $n = 64k$, $n = 256k$, $(k = 2^{10})$, for the last value of n making bitonic sorting faster than Cole's algorithm, and for the first value of n making Cole's algorithm to a faster algorithm. The *exact* numbers, which are shown in the table, of time units consumed by the algorithms for the various problem sizes are *not* especially important. What is important is that the method of algorithm investigation described

Table 4.9: Calculated performance data for the two CREW PRAM implementations.

Algorithm	n	time	# processors
ColeMergeSort	65536 (64k)	4.5×10^4	2.5×10^5
BitonicSort	65536 (64k)	1.2×10^4	3.3×10^4
ColeMergeSort	262144 (256k)	5.1×10^4	1.0×10^6
BitonicSort	262144 (256k)	1.5×10^4	1.3×10^5
ColeMergeSort	2^{67}	196094	5.7×10^{20}
BitonicSort	2^{67}	193620	7.4×10^{19}
ColeMergeSort	2^{68}	199024	1.1×10^{21}
BitonicSort	2^{68}	199365	1.5×10^{20}

in this thesis makes it possible to do such exact calculations.

We see that our straightforward implementation of Batcher's bitonic sorting is faster than the implementation of Cole's parallel merge sort as long as the number of items to be sorted, n, is less than $2^{68} \approx 3.0 \times 10^{20}$. In other words, *for Cole's algorithm to become faster than bitonic sorting, the sorting problem instance must contain close to 1 Giga Tera items!* (10^{21} is about as large as the number of milliseconds since the "Big Bang" [Har87].)[20] This comparison strongly indicates that Batcher's well known and simple $O(\log^2 n)$ time bitonic sorting is faster than Cole's $O(\log n)$ time algorithm *for all practical values of n.* The huge value of n (2^{68}) also gives room for a lot of improvements to the implementation of Cole's algorithm before it beats bitonic sorting for practical problem sizes. There are also good possibilities to improve the implementation of bitonic sorting.

In fact, Cole's algorithm is even less practical than depicted by comparison of execution time. It requires about 8 times as many processors as bitonic sorting, and it has a far more extensive use of the global memory. This is outlined in the following subsection.

[20]The problem size making Cole's algorithm faster than bitonic sorting requires about 1.2×10^{21} processors. If we assume that each processor occupies 5 cubic meters, this corresponds roughly to filling the whole volume of the Earth with processors.

4.4.2.2 Comparison of Cost and Memory Usage

For both algorithms we have derived exact expressions for the execution time and processor requirement as functions of n. Consequently the cost of running each of the two implementations on a problem of size n is known;

$$Cost(\mathit{ColeMergeSort}, n) = T(\mathit{ColeMergeSort}, n) \times \mathit{NoOfProcs}(n) \quad (4.11)$$

where $T(\mathit{ColeMergeSort}, n)$ is given in Equation 4.10 and $\mathit{NoOfProcs}(n)$ is given by Equations 4.9, and 4.6 to 4.8.

For bitonic sorting we have

$$Cost(\mathit{BitonicSort}, n) = T(\mathit{BitonicSort}, n) \times (n/2) \quad (4.12)$$

where $T(\mathit{BitonicSort}, n)$ is given in Equation 4.1.

Using Equation 4.11 and Equation 4.12 it is found that bitonic sorting has a lower cost as long as n is less than 10^{163}, another extremely large number.[21]

This comparison further strengthens the conclusion made above, Cole's algorithm is not a good choice for practical problem sizes.

Global memory access pattern

The CREW PRAM simulator offers the possibility of logging all references to the global memory. Since the global memory is the most unrealistic part of the CREW PRAM model, it may be interesting to monitor how much an algorithm utilises the global memory. An algorithm with an intensive use of the memory may be more difficult to implement on more realistic models without a global memory. Figures 4.25 and 4.26 illustrates the memory usage of the CREW PRAM implementations of Bitonic sorting and Cole's algorithm respectively. Again, it is clearly demonstrated that Bitonic sorting is the best alternative for practical use.

In the figures, plots along a horizontal line depict a memory location accessed by several processors. Plots along a vertical line depict the various memory locations accessed by a single processor.

[21]Imagine filling the whole universe with neutrons such that there is no remaining empty space—the required number of neutrons is estimated to about 10^{128} [Sag81].

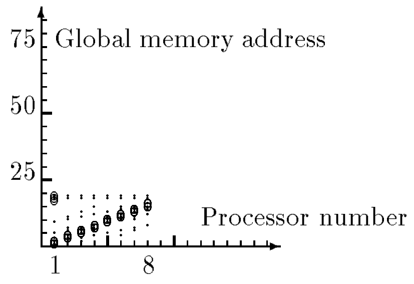


Figure 4.25: Access to the global memory during execution of *BitonicSort* for sorting 16 numbers. · marks a read operation, and ∘ marks a write operation.

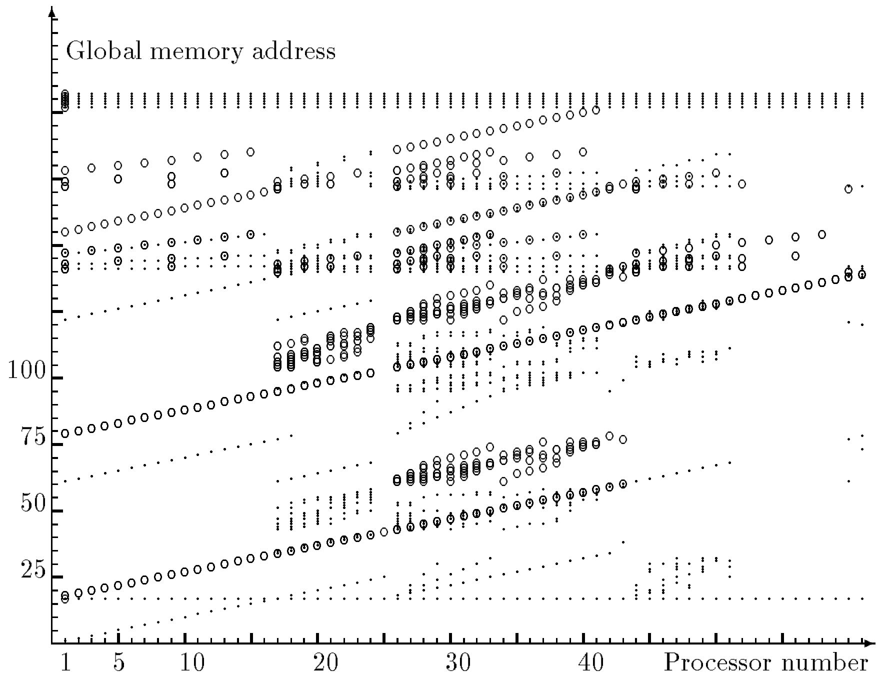


Figure 4.26: Access to the global memory during execution of *ColeMergeSort* for sorting 16 numbers. · marks a read operation, and ∘ marks a write operation.

Chapter 5

Concluding Remarks and Further Work

> "*The lack of symmetry is that the power of a computational model is robust in time, while the notion of technological feasibility is not.* This latter notion is fluid, since technology keeps advancing and offering new opportunities. Therefore, it makes sense to fix the model of computation (after carefully selecting one) and study it, and at the same time look for efficient implementations."
>
> Uzi Vishkin in *PRAM Algorithms: Teach and Preach* [Vis89].

It has been a very interesting job to do the research reported in this thesis. I have found theoretical computer science to be a rich source of fundamental and fascinating results about parallel computing. During the work, new and interesting concepts and issues have continuously "popped up".

5.1 Experience and Contributions

5.1.1 The Gap Between Theory and Practice

The main goal for the work has been to learn about how to evaluate the practical value of parallel algorithms developed within theoretical computer science.

Asymptotically large problem sizes does not occur in practice! By evaluating the performance of *implemented* algorithms for *finite* problem sizes, the effect of the size of the complexity constants becomes visible. My investigation of Cole's algorithm has shown that *Batcher's bitonic sorting algorithm is faster than Cole's parallel merge sort algorithm for all practical values of n.* This is in contrast to the asymptotical behaviour, which says that Cole's algorithm is a factor of $O(\log n)$ time faster.

In my view, it is probably a relation between high descriptional complexity and large complexity constants. However, the computational complexity is not a good indicator for the size of the complexity constants. Recall that Cole's algorithm is considered to be a simple algorithm within the theory community. The majority of the parallel algorithms recently proposed within theoretical computer science has a larger descriptional complexity than Cole's algorithm. Consequently, my investigation of Cole's algorithm has confirmed the belief that promising theoretical algorithms can be of limited practical importance.

This work should not be perceived as criticism of theoretical computer science. In the Introduction, I argued that parallel algorithms from the "theory world", and topics such as complexity theory, will become increasingly important in the future. However, my work has indicated that practitioners should *take results from theoretical computer science with "a grain of salt".* Careful studies should be performed to find out what is hidden by high-level descriptions, and asymptotical analysis.

5.1.2 Evaluating Parallel Algorithms

A new method for evaluating PRAM algorithms?
The proposed method for evaluating PRAM algorithms is based on implementing the algorithms for execution on a CREW PRAM simulator. I have not found the use of this or similar methods described in the literature. The main advantage of the method is that algorithms may be compared for finite problems. As shown by the investigation of Cole's algorithm, such implementations make visible information about performance that is hidden behind the "Big-Oh curtain".

Implementation takes time
I originally planned to investigate 3 - 5 parallel algorithms from theoretical computer science. It was far from clear how much work would be needed to do the implementations. Only Cole's algorithm has been implemented so far. Marianne Hagaseth is currently working on an implementation of

the well known (theoretical) sorting algorithm presented by Shiloach and Vishkin in 1981 [SV81], [Hag91].

For several reasons, *implementing theoretical algorithms takes a lot of time.* First of all, the algorithms are in general rather complex. They are described at a relatively high level, and some "uninteresting" details may have been omitted. However, to be able to implement, you have to know absolutely all details. This makes it necessary to do a thorough and complete study of the algorithm, until all parts are clearly understood. When the algorithm has been clearly described and understood, several implementation decisions remain.

The first implementation of Cole's algorithm

During the detailed study of Cole's algorithm, some missing details were discovered. This made it necessary to do some few completions of the original description (Section 4.2, [Nat90a]). In the development of the CREW PRAM implementation of Cole's algorithm, the most difficult task was to program the dynamic processor allocation in the form of a "sliding pyramid" of processors (Section 4.3.1.3).

To my knowledge, *my implementation of Cole's algorithm is the only implemented sorting algorithm using $O(\log n)$ time with asymptotically optimal cost* [Col90, Rud90, Zag90].[1] This was reported in my paper "*Logarithmic Time Cost Optimal Parallel Sorting is* Not Yet *Fast in Practice!*", presented at the SUPERCOMPUTING'90 conference in November 1990, [Nat90b].

A lot may be learned from medium-sized test runs

Massively parallel algorithms with polylogarithmic running time are often complex. One might think that evaluation of such algorithms would require processing of very large problem instances. So far, this has not been the case. In studying the relatively complex Cole's algorithm, some hundreds of processors and small sized memories have been sufficient to enlighten the main aspects of the algorithm. In many cases, the need for brute force (*i.e.*, huge test runs) may be reduced by the following "working rules":

1. The *size* of the problem instance is used as a *parameter* to the algorithm which is made to solve the problem for all possible problem sizes.

[1]Marcoz Zagha has recently implemented a simplified version of the Reif/Valiant Flashsort algorithm [RV87] on the Connection Machine, however he has expressed that the implementation is far from $O(\log n)$ time [Zag90].

2. *Elaborate testing* is performed on all problem sizes that are within the limitations of the simulator.

3. A detailed *analysis* of the algorithm is performed. The possibility of making such an analysis with a reasonable effort *depends strongly on* the fact that the algorithm is *deterministic* and *synchronous*.

4. The analysis is confirmed with *measurements* from the test cases.

Together, this will often make it possible to use the analytical performance model by extrapolation for problem sizes beyond the limitations of the simulator.

5.1.3 The CREW PRAM Simulator

About the prototype

Several critical design choices were made to reduce the work needed to provide high-level parallel programming and the CREW PRAM simulator. The PIL language was defined to be a simple extension of SIMULA [BDMN79, Poo87], and it was decided to implement the simulator by using the DEMOS simulation package [Bir79]. A program in PIL is translated to "pure" SIMULA which is included in the description of a processor object in the DEMOS model of the simulator. Before execution, the simulator program including the translated PIL program is compiled by the SIMULA compiler (see Section A.1.3).

This is not an efficient solution, but it has several advantages. The translation from PIL to SIMULA is easy, and all the nice features of SIMULA are automatically available in PIL. Perhaps most important, executing the PIL programs as SIMULA programs together with the simulator code makes it possible to use the standard SIMULA source code level debugger [HT87] on the parallel programs and their interaction with the simulator.

Simplicity was given higher priority than efficient execution of the PIL programs. In retrospect, this was a right choice. The bottleneck in investigating Cole's algorithm was the program development time, not the time used to run the program on the simulator. (See the previous section about medium-sized test runs.) In addition, the use of the simulator prototype has given us a lot of ideas about how a more efficient and elegant simulator system should be designed.

Synchronous MIMD programming

The simplicity of the CREW PRAM model combined with the very nice properties of synchronous computations make the development, analysis and debugging of parallel algorithms to a relatively easy task.

Synchronous MIMD programming implies redundant operations, such as compile time padding. However, I believe it is more important to make parallel programming easy, than to utilise every clock period of every processor. Indirectly, easy programming may also imply more efficient programs—because the programmer may get a better overview of all aspects of a complex software system, including performance.

Synchronous MIMD programming, which has been claimed to be easy, should be considered as a by-product of this work. However, it may be one of the most interesting parts for future research. In this context, it was very encouraging to hear the keynote address at the SUPERCOMPUTING'90 conference held by Danny Hillis [Hil90]. Hillis argued that the computer industry producing massively parallel computers is searching for a programming paradigm that combines the advantages of SIMD and MIMD programming.[2] (Similar thoughts have been expressed by Guy Steele [Ste90]).

According to Hillis, it is at present unknown what such a MIMD/SIMD combination would look like. The synchronous MIMD programming style used in the work reported in this thesis is such a combination! To conclude, *synchronous MIMD programming is a very promising paradigm for parallel programming* that should be further investigated.

Use in education

The CREW PRAM simulator system has been used in the courses "Highly Concurrent Algorithms" and "Parallel Algorithms" which are given by the Division of Computer Systems and Telematics (IDT), at the Norwegian Institute of Technology (NTH). It has been used for experimenting with various parallel algorithms, as well as for modelling of systolic arrays and neural networks.

[2]The main advantage of the SIMD paradigm is that the synchronous operation of the processors implies easier programming. MIMD has the advantage of more flexibility and better handling of conditionals.

5.2 Further Work

"How far can hard- and software developments proceed independently? When should they be combined? Parallelism seems to bring these matters to the surface with particular urgency".
Karen Frenkel in [Fre86b].

"What is the right division of labour between programmer, compiler and on-line resource manager? What is the right interface between the system components? One should not assume that the division that proved useful on multiprogrammed uniprocessors is the right one for parallel processing."
Marc Snir in *Parallel Computation Models — Some Useful Questions* [Sni89].

It is my hope to continue working with many of the topics discussed in this thesis. Below I present a short list of natural continuations of the work.

- Continuing the study of parallel algorithms from theoretical computer science, to learn more about their practical value. A very large number of candidate problems and algorithms exist.

- Extend the study on parallel sorting algorithms. Two important alternatives are the Reif/Valiant *flashsort* algorithm [RV87] and the *adaptive bitonic sorting* algorithm described by Bilardi and Nicolau [Bil89]. The flashsort algorithm is a so-called probabilistic algorithm, which sorts in $O(\log n)$ time with high probability. It has a rather high descriptional complexity, but is expected to be very fast [Col90].

- Improving the simulator prototype and the PIL language. Studying what kind of tools and language features would make synchronous parallel programming easier.

- Extending the simulator to the EREW and CRCW variants of the PRAM model. Implementing the possibility of charging a higher cost for accesses to the global memory.

- Defining a more complete language for synchronous MIMD programming, and implementing a proper compiler with compile time padding.

- Leslie Valiant's recent proposal of the BSP model as a bridging model for parallel computation was presented in Section 2.1.3.4. The use of this model raises many questions. Is a PRAM language ideal? How should PRAM programs be mapped to the BSP model? How much of the mapping should be done at compile time, and what should be left to run-time? How is the BSP model best implemented? Much interesting research remains before we know the conditional answers to these questions.

Appendix A

The CREW PRAM Simulator Prototype

"While a PRAM language would be ideal, other styles may be appropriate also."

Leslie Valiant in *A Bridging Model for Parallel Computation* [Val90].

The main motivation for developing the CREW PRAM simulator was to provide a vehicle for evaluation of synchronous CREW PRAM algorithms. It was made for making it possible to implement algorithms in a high level language with convenient methods for expressing parallelism, and for interacting with the simulator to obtain measurement data and statistics.

The most important features for algorithm implementation and evaluation, and the realisation of the simulator prototype, are briefly outlined in Section A.1. A much more detailed description may be found in the User's Guide in Section A.2.

The appendix ends with a description of how systolic arrays and similar synchronous computing structures may be modelled by using the simulator.

A.1 Main Features and System Overview

A.1.1 Program Development

Wyllie [Wyl79] proposed a high level pseudo code notation called parallel pidgin algol for expressing algorithms for the PRAM model. Inspired by this work, a notation called PIL has been defined. The PIL notation (Parallel Intermediate Language) is based on SIMULA [BDMN79, Poo87][1] with a small set of extensions

[1]SIMULA is a high-level object-oriented programming language developed in Norway. It was made available in 1967, but is still modern!

which is necessary for making it into a language well suited for expressing synchronous MIMD parallel programs. Nearly all features found in SIMULA may be used in a PIL program.

The main features for program development are:

- *Processor allocation.* A PIL program may contain statements for allocating a given number of processors and binding these to a *processor set.* A processor set is a variable used to "hold" these processors, and may be used in processor activations.

- *Processor activation.* A processor activation specifies that all processors in a given processor set are to execute a specific part of the PIL program.

- *Using global memory.* The simulator contains procedures for allocating and accessing global memory. The procedures ensures true CREW PRAM parallelism and detection of write conflicts.

- *Program tracing.* A set of procedures for producing program tracing in a compact manner is provided.

- `SYNC` and `CHECK`. *This is two built-in procedures which have shown to be very helpful during the top-down implementation of complicated algorithms.* `SYNC` causes the calling processors to synchronise so that they all leave the statement simultaneously. `SYNC` has been implemented as a "binary tree computation" using only $O(\log n)$ time (see Section 3.2.5.1). `CHECK` checks whether the processors are synchronous at the place of the call in the program, *i.e.* that the processors are doing the same call to `CHECK` simultaneously.

- *File inclusion, conditional compilation and macros.* This is implemented by preprocessing the PIL program by the standard C preprocessor `cpp`.

Perhaps the greatest advantage of defining the PIL notation as an extension of SIMULA is that it implies the availability of the SIMULA debugger `simdeb` [HT87]. This is a very powerful and flexible debugger that operates on the source code level. It is easy to use for debugging of *parallel* PIL programs. As an example, consider the following command:

```
at 999 if p = 1 AND Time > 2000 THEN stop
```

This command specifies that the control should be given to the debugger when source line number 999 is executed by processor number 1 if the simulated time has exceeded 2000.

A.1.2 Measuring

The simulator provides the following features for producing data necessary for doing algorithm evaluation:

- *Global clock.* The synchronous operation of the CREW PRAM implies that one global time concept is valid for all the processors. The global clock may be read and reset. The default output from the simulator gives information about the exact time for all events which produce any form of output.
- *Specifying time consumption.* In the current version of the simulator, the time used on local computations inside each processor must be explicitly given in the PIL program by the user. This makes it possible for the user to choose an appropriate level of "granularity" for the time modelling. It also implies the advantage that the programmer is free to assume any particular instruction set or other way of representing time consumption. However, future versions of the CREW PRAM simulator system should ideally contain the possibility of automatic modelling of the time consumption, performed by the compiler.
- *Processor and global memory requirement.* The number of processors used in the algorithm is explicitly given in the PIL program, and may therefore easily be reported together with other performance data. The amount of global memory used may be obtained by reading the user stack pointer.
- *Global memory access statistics.* The simulator counts and reports the number of reads and the number of writes to the global memory. Later versions may contain more advanced statistics.
- *DEMOS*[2] *data collection facilities.* The general and flexible data collection facilities provided in DEMOS are available; `COUNT` may be used to record events, `TALLY` may be used to record time independent variables—with various statistics (mean, estimated standard deviation etc.) maintained over the samples, and `HISTOGRAM` provides a convenient way of displaying measurement data.

Figure A.1 shows the general format of a main program (PIL) for testing an algorithm on a series of problem instances. When generating problem instances, the random drawing facilities in DEMOS which sample from different statistical distributions may be very useful.

Functions such as processor allocation and processor resynchronisation are realised in the simulator by simulating real CREW PRAM algorithms for these tasks—to obtain realistic modelling of the time consumption.

A.1.3 A Brief System Overview

When developing the CREW PRAM simulator prototype it was a major goal to make a small but useful, and easily extensible system. This had to be done within a few man months—so it would be impossible to do all from scratch.

[2]DEMOS is a nice and powerful package for discrete event simulation written in SIMULA [Bir79].

```
...
PROCESSOR_SET ProcessorSet;
INTEGER n;
ADDRESS addr;
...
FOR n := 2 TO 512 DO
BEGIN
   addr := GenerateProblemInstance(n);
   ... the problem instance is now placed in global memory
   ClockReset;
   ... Start of evaluated algorithm
   ... initial sequential part of algorithm
   ASSIGN n PROCESSORS TO ProcessorSet;

   FOR_EACH PROCESSOR IN ProcessorSet;
      ... PIL statements executed in parallel by
          all the processors in ProcessorSet
   END_FOR_EACH PROCESSOR;
   ... End of evaluated algorithm

   TimeUsed := ClockRead;
   ... sequential code for checking and/or reporting the
       results obtained by running the algorithm
   ... sequential code for gathering and reporting performance
       data from the last test run
END_FOR loop doing several test runs;

... sequential code for reporting statistics about all the test runs
```

Figure A.1: General format of PIL program for running a parallel algorithm on several problem instances. In this case the algorithm is assumed to use one set of n processors where n is the size of the problem instance.

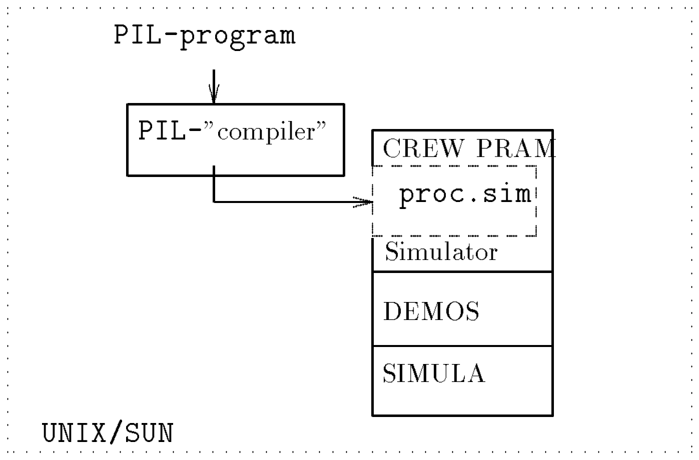


Figure A.2: CREW PRAM Simulator — implementation overview.

The main implementation principle used in the simulator is outlined in Figure A.2. The CREW PRAM simulator is written in SIMULA [BDMN79], [Poo87], [DH87] using the excellent DEMOS discrete event simulation package [Bir79]. The PIL program is "compiled"[3] to SIMULA/DEMOS and included in the file `proc.sim` which is part of the simulator source code. `proc.sim` describes the behaviour of each CREW PRAM processor. Thus, during algorithm simulation, each processor object (in the SIMULA sense) executes its own copy of the PIL program.

A great advantage implied by this implementation strategy is that all the powerful features of the SIMULA programming language and the DEMOS package are available in the PIL program. Further the source level SIMULA debugger `simdeb` [HT87] may be used for studying and interacting with the parallel PIL program under execution.

This is certainly a resource demanding implementation. However, experience have shown, as was described in Section 5.1.2, that the prototype is able to simulate large enough problem instances to unveil most interesting aspects of the evaluated algorithm.

During the development of the CREW PRAM implementation of Cole's algorithm, all, except one, errors were found by testing the program on problem instances consisting of only 16 numbers. The last error was found by testing the implementation on 32 number, however, the same error was also visible when sorting only 8 numbers.

Executing Cole's algorithm on 16 numbers requires 61 CREW PRAM processors and takes about 60 CPU seconds on a SUN 3/280. 986 CREW PRAM processors are needed to sort 256 numbers, and require about 4 CPU hours. Bitonic sorting is simpler and requires fewer processors, which make the simulations faster. 16 numbers are sorted in 4 CPU seconds, and 4096 numbers (corresponding to 2048 CREW PRAM processors) are sorted in about 3 CPU hours (SUN 3/280).

Implementing the simulator in SIMULA/DEMOS implies an open-ended easily extensible system. The CREW PRAM simulator may easily be modified to an *EREW* PRAM simulator.

A.2 User's Guide

This section describes how to use the CREW PRAM simulator as it currently is installed at the Division of Computer Systems and Telematics (IDT), The Norwegian Institute of Technology (NTH).

A.2.1 Introduction

The CREW PRAM simulator includes a simple programming environment that makes it convenient to develop and experiment with synchronous MIMD parallel

[3]This is a simple transformation done by a pipeline of filters written in 'gawk' (GNU awk).

algorithms. Parallel algorithms are in general more difficult to develop, understand, and test than sequential algorithms. However, the property of synchronous operation combined with features in the simulator removes a lot of these difficulties.

The parallel algorithms may be written in a high-level SIMULA like language. A powerful symbolic debugger may be used on the parallel programs.

A.2.1.1 Parallel Algorithms, CREW PRAM, PPP and PIL

When studying parallel algorithms it is desirable to use a machine model that is simple and general—so that one may concentrate on the algorithms and not the details of a specific machine architecture. The CREW PRAM satisfies this requirement.

Further, when expressing algorithms it has become practice to use so called pseudo code. In Section 3.2.2 a notation called PPP (Parallel Pseudo Pascal) was presented.

Just as other kinds of pseudo code, PPP may not yet be compiled into executable code. It is therefore needed a notation at a lower level. The motivation behind the work that has resulted in the CREW PRAM simulator is a study of a special kind of parallel algorithms—not to write a compiler. Therefore, the programming notation which has been adopted is a compromise between the following two requirements; 1) It should be as similar to PPP as possible, 2) It must be easy to transform into a format that may be executed with the simulator. The notation is called PIL, which is short for Parallel Intermediate Language. It is described in Section A.2.3.1.

A.2.1.2 System Requirements and Installation

The CREW PRAM simulator may be used on most of IDT's SUN computers running the UNIX[4] operating system.Note that the `run` command very well may be executed locally on a SUN workstation—this reduces your chance of becoming unpopular among the other users.

We assume that you are using the UNIX command interpreter which is called `csh`. First, you must add the following line at the end of your `.cshrc` file in your login directory.

```
set path = (../cmd $path)
```

Then make sure that your current directory is your login directory, and type

```
source .cshrc
```

to make the change active.

Use `cd` to move to a directory where you want to install the version 1.1 of the CREW PRAM simulator. Type the following two commands, which will install the software in a new directory called `ver1.1`.

[4]UNIX is a trademark of AT&T.

```
ASSIGN 2 PROCESSORS TO ProcSetA;
FOR_EACH PROCESSOR IN ProcSetA
  Use(1);
  T_TO("Hello World!");
END_FOR_EACH PROCESSOR;
```

```
...
@<<<   TIME  103-----:
@Hello World!
@Hello World!
@<<<   TIME  104-----:
...
```

Figure A.3: Part of simple demo program (left), and part of produced output (right).

```
mkdir ver1.1
/sigyn/home/lasse/dring/releases/version1.1/cmd/install1.1 ver1.1
```

A.2.2 Getting Started

Without going into details, this section shows how a very simple example program may be compiled and executed with the simulator.

The current installation of the CREW PRAM simulator has a directory called `pil` which contains a collection of PIL programs which illustrates various aspects of the CREW PRAM simulator and parallel algorithms. Consider the very simple parallel algorithm on the file `demo1.pil`. The main part of this program and its output is shown in Figure A.3.

Be sure that your current directory is `src`. To compile the PIL program stored in the file called `demo1.pil` under the `pil` directory, give the command:

```
pilc demo1
```

After compiling, you must use the `link` command. Just type `link`. Now, you are ready to execute the program, type `run`.

A.2.3 Carrying On

In this section we describe most of the features in the CREW PRAM simulator, the related tools, and the PIL language.

As an example we will use a simple PIL program: Odd-Even Transposition Sort is one of the simplest parallel algorithms for sorting. Descriptions of the algorithm may be found in [Akl85] pp. 41–44 or [Qui87] pp. 88–89. A PIL program that implements this algorithm may be found on the file `oes1.pil` under the `pil` directory.

Make a copy of `oes1.pil` to another filename so you do not destroy the original version. In the following, we assume that your copy was named `oddeven.pil`.

A.2.3.1 The PIL Language—Background

The PIL (Parallel Intermediate Language) notation is based on SIMULA with a small set of extensions which is necessary for making it into a language well suited for expressing synchronous MIMD parallel programs.

Consequently, nearly all features found in SIMULA may be used in a PIL program. The main exceptions to this are mentioned in this appendix. The CREW PRAM simulator is implemented by good help of the simulation package DEMOS. DEMOS offers a lot of powerful features which all are made available through the PIL language. In the context of analysing and measuring parallel algorithms—a large set of procedures for gathering and reporting statistics may be very helpful.

A recent book on SIMULA is *An Introduction to Programming in SIMULA* by R. J. Pooley [Poo87], an older one is *SIMULA BEGIN* written by Graham Birtwistle and the three Norwegians Ole-Johan Dahl, Bjørn Myhrhaug and Kristen Nygaard who invented SIMULA [BDMN79].

DEMOS is written in SIMULA, and is described in the book *DEMOS—A System for Discrete Event Modelling on Simula* written by Graham Birtwistle [Bir79]. This book also gives a small introduction to SIMULA, sufficient for making the reader able to utilise all the powerful features in DEMOS.

The current implementation of the CREW PRAM simulator is based on the SIMULA system offered by Lund Software House AB, Sweden. Specific details of this version of the SIMULA language may be found in [DH87, HT87].

A.2.3.2 Including Files, Macros and Conditional Compiling

Three useful features are made available by letting the PIL compiler run the standard UNIX C preprocessor on the input file. They are explained in the following.

Further documentation of these features may be found in [KR78], or just type `man cpp` on any UNIX terminal. Note that it is *not* possible to pass parameters to the `cpp` command in this version of `pilc`.

`#include`
The reader which is familiar with the C programming language will recognise the syntax used for including files. `oddeven.pil` includes three files—`sorting`, `oes.var` and `oes.proc`. It is good practice to split your algorithm in several logical units. When the PIL compiler `pilc` encounters the line

```
#include<sorting>
```

it will look for a file named `sorting` in the `pil` directory. If it is found, the contents are copied into the program at the place of the `#include` statement.

`#define`
`oddeven.pil` contains one macro named `SHOWodd` which is defined by the `#define`

command. This macro is simply used to make a shorter name on the (more precise) procedure call `Let1ShowArray("Odd phase done");`.[5]

Macros *may* be used to make programs more readable—however complicated macros and especially macros with parameters should be used with care. Macros are in general dangerous.

A few names, `ADDRESS`, `CONSTANT` and `INFTY` have been defined as macros in a file, called `pil.h` which is included by the PIL compiler. These names are nothing more that syntactic sugar.

`#ifdef, #else, #endif` etc.

It is often desirable to operate with several versions of a program, but with only slight changes between two or more versions. Two versions may be stored as two (very similar) files. However, it is normally much better to let the difference be visible inside one copy of the program. Conditional compiling makes this possible.

Study the lines beginning with a `#` around `SHOWodd` and `SHOWeven` in our sample file. Insert `%` and one space in the front of the line `#define VERBOSE`, recompile, link and run—and see what happens. (SIMULA lines beginning with `%` followed by a space are interpreted as comments by the SIMULA compiler).

Excessive use of this feature may make the programs unreadable.

A.2.3.3 Program Structure, Procedures and Variable Declarations

When you start the CREW PRAM simulator, one CREW PRAM processor will automatically start execution at top of the CREW PRAM program. This processor is sometimes called the *master processor*, even though it is equal to all the other CREW PRAM processors. If you do not allocate and activate processors (discussed in the following two subsections), the program will be executed as a sequential program by this single processor.

Procedures

As for sequential programs, it is important to structure your program by use of procedures. Procedures may be nested. Top down development and a hierarchy of procedures is warmly recommended since PIL programs are not easier to develop than traditional sequential programs.

Procedures in PIL have the same format as in SIMULA with the following exception. In the current version, *all* procedures[6] must begin with a `BEGIN` statement and end with a statement of the form:

`END_PROCEDURE` *ProcedureName*;

[5]This procedure is defined in the file `oes.proc`. More on procedures in Section A.2.3.3.

[6]In standard SIMULA procedures may consist of a single statement not enclosed by the SIMULA keywords `BEGIN` and `END`. This is *not* allowed in PIL.

ProcedureName may be any text, but it is recommended to use the name of the procedure.

Variables—make them as local as possible

Declaration of processor local variables follow the SIMULA syntax. *All* variables which are declared at the outermost level in the PIL program will be local to *each* processor. To illustrate this, consider a program that consists of two logically different parts, A and B. Part A is executed by processor set *A* and part B is executed by the *B* processors. Part A does only use variable *var-A* and B uses only *var-B*. In the current version, all the processors in processor set *A and B* will have its own copy of *both var-A* and *var-B*, even though the processors in *B* do not use *var-A* and vice versa.[7]

This is not an ideal situation. If the programmer in the code executed by the *A* processors erroneously uses *var-B* no error message will occur—as one should expect from a proper compiler. The *A* processors will use its own local copy which will contain the value zero, at least the first time it is (erroneously) used.

In large programs, with several kinds of processors and many variables, this oddity makes the program less readable with respect to variable usage.[8] In the current implementation, it also makes the data part of each processor object unnecessary large—which in turn slows down the simulation.

However, this problem may to a large extent be alleviated by making variables local where it is possible. Remember that variables in SIMULA (and therefore also in PIL) may be declared as local for an arbitrary block, in addition to local inside a procedure.

As an example, compare `oes1.pil` with the file `oesNew2.pil` shown in Figure 3.21 at page 95. In the latter program, both procedures and variables have been moved inside the parallel part (`FOR_EACH PROCESSOR ...`) of the program. This programming style is generally recommended.

Variables in the global memory

Global variables, *i.e.* variables placed in the CREW PRAM global memory are rather restricted in the current version of PIL. Section A.2.3.6 explains how integer locations in the global memory are allocated and accessed by using the PIL `READ` and `WRITE` statements. These statements require the specification of a global memory *address*. Variables used to hold such addresses should be declared as of type `ADDRESS`.

[7]The reason for this odd feature is that the implementation of the simulator is strongly based on letting each processor (as a SIMULA object) execute its own copy of the whole PIL program.

[8]In a way, it gives the same problems as traditionally encountered in large sequential programs with a lot of global variables—it is difficult to read from the code which variables are used or may be used in various parts of the code.

A.2.3.4 Processor Allocation

All *parallel* PIL programs must contain at least one statement for processor allocation. The format is

```
ASSIGN Number PROCESSORS TO ProcSet;
```

This statement allocates *Number* new processors in a processor group or set which is given the name *ProcSet*. The term *processor set* is used to denote such a collection of processors. The master processor mentioned above is a processor set containing only one processor, and is automatically allocated when starting the simulator. *Number* may be any expression resulting in an integer. Note however that in the current version this expression must be written as one text-string without any whitespace[9] characters. *ProcSet* is the name of a variable which must be declared as type `PROCESSOR_SET`. See the declaration of the variable `ProcSet1` in the file `oes.var` included from `oddeven.pil`.

Request for processor allocation can only be issued by the single processor (master processor) which executes the outermost level of the main program. However, the allocation is done by the allocated processors in cooperation in a binary tree fashion. As a result, allocating n processors takes $O(\log n)$ time units. (See Section 3.2.5.1.)

Any number of processors may be allocated to a processor set, and any number of processor sets may be allocated and used in a program.

A.2.3.5 Processor Activation

When a set of processors has been allocated, they may be specified to execute code in parallel by use of so called processor activation. *Processor activations must be placed at the outermost level of the main program*—with the exception that they may be done inside (eventually) nested `FOR` and/or `WHILE` loops. Currently, processor activation may not be placed inside a (general) `BEGIN` ...`END` block. For further details, study the file `LoopTest.pil` in the `pil` directory. The format is

```
FOR_EACH PROCESSOR IN ProcSet
    AnyCode
END_FOR_EACH PROCESSOR;
```

AnyCode will be executed by all the processors in *ProcSet* in parallel—*and it is important to remember that each of these processors will operate on its own set of variables*. The processors in *ProcSet* are numbered $1, 2, 3, \ldots$, and each processor knows its own number through the local variable `p`. This variable should not be changed by the processor (PIL program).

The single processor (master) which is dedicated for executing the outermost level of the main program will "wait outside" while *AnyCode* is executed.

[9] Whitespace is common UNIX terminology for space (blank), tab, and newline.

AnyCode may contain variable and procedure declarations if it is written as a SIMULA block (`BEGIN` ... `END`). If it does *not* contain declarations, it may be written as a list of statements without the enclosing `BEGIN` ... `END`. The `END_FOR_EACH PROCESSOR` statement checks that the processors are synchronous when they leave *AnyCode*. If not, a warning "`SYNCHRONY LOST`" is given, see page 188.

Nesting of processor activations is not allowed. Currently, only one processor set may be active at the same time.

A.2.3.6 Using the Global Memory

All processors in a CREW PRAM are connected to a global memory. This memory must be used for all kinds of communication between the processors. The global memory is simply a large array of integers. On the simulator, the global memory has *fixed* size and may be regarded as consisting of two parts—the *user stack* and the *system stack*. The system stack is used to store the processor set data structures. The *user stack* is used to store global variables allocated dynamically in the PIL program. It grows towards higher addresses.

The number of locations read/written from/to the global memory is measured by the simulator and reported at end of a PIL program execution.

Allocating memory

Global memory must be allocated explicitly before it can be used, see the procedure `Mem.UserMalloc` in Section A.2.3.8. In our sample program, memory is allocated in the procedure `GenerateSortingInstance` in the file `sorting`.

Local memory should *not* be allocated explicitly in a PIL program. Just declare your variables, and the (SIMULA) system does the remaining work.

Accessing global memory

Any processor may read from any location in the global memory at any time. The format is:

`READ` *VarName* `FROM` *AddressExpression*;

The contents of the location in the global memory given by *AddressExpression* is read and placed in the integer variable called *VarName*. *AddressExpression* may be an arbitrary integer expression. In the current version, *AddressExpression* and *VarName* must not contain any whitespace characters. The PIL keyword `READ` must be the first text symbol on the line, and the whole statement must be written on a single line (but it may exceed 80 characters). In the current version, a `READ` statement takes one time unit regardless of the complexity of *AddressExpression*.

Any processor may write any value to any location in the global memory at any time. But, if two or more processors at the same time unit writes to the same global memory location, a *write conflict* will occur. This is an illegal operation on a CREW PRAM, and the CREW PRAM simulator will give an appropriate error message, and stop. The format for writing to the global memory is:

```
WRITE Value TO AddressExpression;
```

Value may be any integer expression. Note that for every time unit, all writes to global memory occur *after* all reads from the memory. (See CREW PRAM property 2 at page 64.) In the current version, a WRITE statement takes one time unit regardless of the complexity of *AddressExpression*. The syntax of the WRITE statement is similar to the READ statement as described above.

Just as standard SIMULA statements, these statements must not end with a ; if they are placed just before the SIMULA keyword `ELSE`.

A.2.3.7 FOR and WHILE Loops

A `FOR` loop should be written according to the following format:

```
FOR LoopVar := FirstVal TO LastVal DO
BEGIN
    AnyCode
END_FOR;
```

The keyword `TO` is used instead of the corresponding `STEP 1 UNTIL` of SIMULA to make PIL closer to the PPP syntax. The whole `FOR` ...`DO` must be contained in a single line. In the current version, *LoopVar*, *FirstVal* and *LastVal* must be written as one text-string without any whitespace characters, and the assignment operator `:=` must be enclosed by spaces.

`WHILE` *loops* have a very similar form:

```
WHILE LoopCondition DO
BEGIN
    AnyCode
END_WHILE;
```

The *LoopCondition* may contain whitespace characters, however the whole `WHILE` ...`DO` statement must be contained in a single line, and `DO` must be the last symbol on that line.

A.2.3.8 Built-In Procedures and Variables

The CREW PRAM simulator offers various built-in procedures and variables which will be used in most PIL programs.

Interaction with the simulator

- `Use(`*TimeUnits*`);`
  This procedure is used to represent time used by the calling processor. *TimeUnits* is an arbitrary integer expression.

- `Wait(`*TimeUnits*`);`
  Does exactly the same as `Use`, but should be used to make it possible to distinguish idle time from active work.

- *IntegerVar* `:= ClockRead;`
  `ClockRead` returns the current time as an integer value. Remember that the synchronous property of a CREW PRAM implies that this clock may be perceived as one global clock which always is correct for all the processors.

- `ClockReset;`
  This call sets the clock to zero. It is typically called after having done uninteresting initialisation work etc., and just before the computation which should be measured starts.

- `SYNC;`
  This procedure makes all the processors in the active processor set to synchronise so that they all will return from the call to `SYNC` simultaneously. This explicit resynchronisation is implemented as a "binary tree computation" with $O(\log n)$ time consumption as outlined in Section 3.2.5.1. Its use together with the `CHECK` procedure in a recommended top down approach for CREW PRAM programming is discussed in Section 3.4 In the current version of the simulator, it is assumed that `SYNC` is called by *all* the processors in the *active* processor set.

- `CHECK;`
  Checks whether *all* the processors in the *active* processor set are synchronous at the place in the program where `CHECK` is called. See also the description of procedure `SYNC` above.

- `Warning(`*Text*`);`
  Prints `''<<< WARNING >>>''` followed by *Text*. The program execution continues.

- `Error(`*Text*`);`
  Prints `''<<< ERROR >>>''` followed by *Text* and stops the program execution.

Using the global memory

See also Section A.2.3.6.

- *AddressVar* `:= Mem.UserMalloc(`*IntegerExpr*`);`
  Allocates *IntegerExpr* number of consecutive memory locations on top of the user stack in the global memory. The address of the first location is returned.

- `Mem.UserFree(`*IntegerExpr*`);`
  Frees *IntegerExpr* number of consecutive memory locations from top of the user stack in the global memory.
- `ArrayPrint(`*StartAddressExpression, Size*`);`
  Prints *Size* consecutive locations in global memory starting at, and including *StartAddressExpression*.
- `PUSH(`*IntegerExpr*`);`
  Allocates a new location on top of the user stack and writes the value of *IntegerExpr* in that location.
- *IntegerVar* `:= POP;`
  `POP` returns the value stored in the location on top of the user stack and frees the location.
- `UserStackPtr`
  This is a globally accessible variable of type `INTEGER` which points to the next *free* location on the user stack.
- `Mem.N_Reads` and `Mem.N_Writes`
  These are DEMOS objects of type `COUNT` which measures the number of `READ` operations and `WRITE` operations to the global memory. The `READ` count may be reset by inserting `Mem.N_Reads.RESET;` in your PIL program, and its current value may be reported by `Mem.N_Writes.REPORT;`. Similarly for `WRITE` operations.

The procedures `PUSH` and `POP` may be used together with the `UserStackPtr` to pass values from a single processor to a large number of processors in a very efficient way. See the file `test7.pil` for a small example of how this may be done to implement "parallel parameter passing", or study the main program and the procedure `SetUp` in the file `systolicdemo1.pil` (explained in Section A.3.2) for a more realistic example.

Generating random data and collecting statistics

A set of global variables has been included in the CREW PRAM simulator to make it easy to use the *DEMOS random number genarators* and the DEMOS facilities for *collecting statistical data*. These global variables make it possible to create the specific DEMOS objects when they are needed (probably done by the master processor), and to use them by any processor without interfering with the simulator time or other quantities that are measured by the simulator.

These variables are of (SIMULA) type `REF(`*DemosClassName*`)` and are named `Global`*DemosClassName*`(`*IndexVal*`)`. *DemosClassName* may be one of `COUNT`, `TALLY` or `HISTOGRAM` with *IndexVal* in the range 1..16, or one of `CONSTANT`, `NORMAL`, `NEGEXP`, `UNIFORM`, `ERLANG`, `EMPIRICAL`, `RANDINT`, `POISSON` or `DRAW` with *IndexVal* in the range 1..4. An extensive example is found in the file `sync6.pil` in the `pil` directory. Consult also the DEMOS book [Bir79].

Tracing and output

Some programmers have the opinion that making code for producing output in SIMULA requires a lot of typing. Also, when debugging programs it is convenient to be able to trace out values of variables during the program execution. Such program tracing should be possible to turn off without the need for removing the code which produces the tracing. It may be useful to keep it also *after* you *believe* you have done your last change to the program. If a later modification introduces a new bug, you will probably have good use for the "old" tracing code.

The CREW PRAM simulator contains a set of simple procedures which may be a good help in making this kind of tracing. The routines may also very well be used for "normal" output from the program. All these procedures starts with `T_`.

- `T_Off;`
  Turns tracing off. Tracing procedures called after this procedure will do nothing.
- `T_On;`
  Turns tracing on. `T_On` and `T_Off` operate on a global flag (`BOOLEAN T_Flag`) which controls tracing for all processors.
- `T_ThisIs;`
  Prints out the variable name of the processor set and the processor number for the processor performing this procedure. This is very useful if you have lost control over your processors.
- `T_Time;`
  Reports the current simulation time.
- `T_TITITO(`*Text1*`, `*IntegerExpr1*`, `*Text2*`, `*IntegerExpr2*`, `*Text3*`);`
  This is the most complicated of a set of similar procedures used to trace out values from the program in a compact manner.
- `T_TITIO, T_TITI, T_TITO, T_TIO, T_TBO, T_TI, T_TO, T_T`
  These procedures are all similar to `T_TITITO`. The part of the name following the underscore character is intended to be a short way to specify the parameters which must be given to the procedure. `T` is short for Text, `I` for Integer, and `B` for Boolean. A procedure with name ending with `O` calls `OUTIMAGE`[10] before it returns.

All lines in the simulator output produced by the "`T_` procedures" start with a `@` in the first column, making them easy to filter away from the other output.

Miscellaneous

- `IsOdd(`*IntegerExpr*`)`
  Returns `TRUE` if *IntegerExpr* evaluates to an odd number.

[10] I/O in SIMULA is buffered, `OUTIMAGE` empties the output buffer.

- `IsEven(`*IntegerExpr*`)`
  Returns `TRUE` if *IntegerExpr* evaluates to an even number.

- `log2(`*IntegerExpr*`)`
  Returns the value of $\log_2(IntegerExpr)$.

- `power2(`*IntegerExpr*`)`
  Returns the value of $2^{IntegerExpr}$.

A.2.4 The Development Cycle

It is now time to give some more details of the *edit—compile— link—run cycle.*

A.2.4.1 `pilc`

The PIL compiler `pilc` consists of two main phases. The first phase produces a SIMULA file, and the second phase compiles this file together with the source code for the simulator. The first phase is implemented as a UNIX pipeline, whose most important parts are the C preprocessor `cpp` and a filter called `PILfix`[11]. These interior details are mentioned here to describe why error messages from the `pilc` command may come from at least three main sources. This will now be demonstrated by introducing some small errors in our sample file `oddeven.pil`

`cpp` errors

Insert a space before the `s` in `sorting` in the `#include` statement at the beginning of the file, and try to compile the file with `pilc`. The output will then contain a line looking like:

```
tmp.ppl: 41: Can't find include file  sorting
```

`tmp.ppl` is a temporary file used by `pilc`. `41` is the line number[12] in that file. Few "early" error messages like this come alone. In our example we also get an error message from the SIMULA compiler. In general, it is seldom worthwhile to invest to much effort in understanding the subsequent error messages.

When encountering error messages from `cpp` remember that `man cpp` is available on your terminal.

Before you proceed, remove the error introduced above.

[11] `PILfix` does the main work of translating PIL specific features into SIMULA code. `PILfix` itself is a pipeline of three simple filters—all written in the pattern scanning and processing language gawk (GNU awk).

[12] This, and all line numbers in later examples are very dependent on the actual version of the simulator and the example file you are using. Please do not expect them to match with what you get on your screen.

`PILfix` errors

During the progress of the compilation `PILfix` produces an indented listing of the procedures found in the program. This is valuable information when looking for errors in the block structure of the program.

In the current version, not all syntax errors that may occur in PIL programs are found and reported by `PILfix`. However, most errors are detected in the next phase—the SIMULA compiler.

Insert a space after the `n` in the `ASSIGN` statement of `oddeven.pil`, and compile. We get the following error message:

```
PILfix: ERROR (at *internal* line no 91)
PIL line was:   ASSIGN n -1 PROCESSORS TO ProcSet1;
Error message: ASSIGN statement --- improper syntax
```

Other errors found by `PILfix` are demonstrated by `test9.pil` and `test10.pil` in the `pil` directory.

`simula` errors

`simula` is the name of the SIMULA compiler [DH87]. It is run on the simulator source files (in the `src` directory), which includes the file `PILalg.sim` which is produced by the (sub-)phases discussed above. It is seldom necessary for the PIL programmer to look at `PILalg.sim`.

Most error messages from `simula` are self-explanatory and will not be discussed here. However, there is one kind of error which often occurs in SIMULA and PIL programs that may produce a frightening amount of error messages. This type of error is often called *error in block structure*. First, let us demonstrate how such an error unwillingly may be introduced in your program.

The traditional format for comments in SIMULA is the keyword `COMMENT` followed by any amount of lines until the comment is ended with a `;`.[13] If you forget to terminate the comment by `;` the comment will (del)eat all code up till the next `;`. This *may* cause an error message, but may also result in a faulty program where important parts are silently omitted.

In our sample file, remove the `;` at the end of the comment just before the statement `END_FOR;` This will produce a lot of error messages referring to names in other files than your program (`PILalg.sim`).

```
*** Error in file: util.sim  <at the beginning of the listing>
...
Error (5). Too few 'end's - extra end inserted    <at the end>
```

The file name `util.sim` refers to one of the CREW PRAM simulator source files.

[13]The SIMULA version used by the CREW PRAM simulator allows `!` instead of `COMMENT`.

Whenever this kind of error message occur the reason is probably an *error in the block structure* in the SIMULA (PIL) program. In this case the last part of the error listing tells the reason. This does not generally happen.[14]

The following habit is warmly recommended (for all block oriented languages) to avoid this kind of errors: When typing a comment of the form `COMMENT` ... `;` or `!` ... `;`, or when typing a `BEGIN` ... `END` structure—always type the "termination part" of the syntax before you type the insides.

How to map line numbers given by the SIMULA system to line numbers in your PIL algorithm is discussed under the `run` command below.

A.2.4.2 `link`

Since the current version of the CREW PRAM simulator compiles the PIL program together with the simulator as one source file there is very seldom that you get errors from the `link` command that have not been reported by the compiler.

A.2.4.3 `run`

SIMULA has a powerful run time system. The PIL programmer takes advantage of this. Together with a run time error message, the source line number where the fail occurred will be printed.

Line numbers

The line numbers in error messages from the SIMULA compiler or run time system will not match the line numbers in the PIL program. The reason is that the PIL program is translated to a larger SIMULA program which is included in the simulator code.

However, it is still easy to find the PIL line corresponding to a SIMULA line. The file `pram.lis` in the `src` directory contains the listing produced by `pilc` and corresponding to the code executed by `run`. Find the line number[15] in this file where the error has occurred. In most cases you will by inspection of `pram.lis` understand where the corresponding line in the PIL program is. For large programs, it may be practical to use an editor with two buffers—one containing your PIL program and one containing `pram.lis`. Then the editor will probably do the pattern matching faster and more reliable.

Error messages from the simulator

We will now shortly explain run time error messages from the simulator.

[14] A "brute force" method for locating such errors is a systematic and repeated exclusion of parts of the program followed by recompilation. The `#include` statement may be helpful in doing this.

[15] There are in fact two columns with line numbers in this listing, the one to the right is the right one for this purpose.

- `WRITE CONFLICT`
  This occurs if two or more processors perform a write operation on the same location (address) in the global memory. The CREW PRAM simulator will stop the program at the end of the time unit when the conflict occurred. See the example file `test12.pil`.
- `SYNCHRONY LOST`
  This is given as a warning from the simulator when the `CHECK` procedure detects that not all processors in the active processor set are synchronous. The SIMULA source line number where `CHECK` was called is included in the message. An example is given in `test13.pil`. Note that the `CHECK` procedure is called as part of the `END_FOR_EACH PROCESSOR` statement.
- `insufficient memory`
  The CREW PRAM simulator implementation is based on allocating memory dynamically as storage is needed during the execution. When running PIL programs using a large number of processors the following message may appear:

  ```
  Runtime Error at line: ... Block at line: ...
  No storage freed by garbage collector
  ```

  The lower limit for the number of processors giving this run time error message may be made larger by using the `m` option for the SIMULA compiler. Consult [DH87] and make your own version of the `pilc` command (but do *not* forget that you are using a multi-user system!).

A.2.5 Debugging PIL Programs Using `simdeb`

Perhaps the greatest advantage of executing the PIL program as a SIMULA program is that it implies the availability of the SIMULA debugger `simdeb` [HT87]. Notice that the debugger operates at the SIMULA source code level.

To help the PIL programmer getting started with using the debugger, we will now give some examples of commands that have proven useful. The various commands may be combined as shown in some of the examples.

The SIMULA debugger is started on your last compiled and linked PIL program by issuing the command `debug`.

1. `help`
   This command produces a listing of the available debugger commands.
2. `help output`
   This is the format for obtaining more information about a specific command, in this case the `output` command.
3. `proceed`
   Starts execution of the program.

4. `at` *LineNo* `stop`
   The `at` command is used to define that some debugger actions should take place when the program executes the specified source line number *LineNo*. In this case we want the program to stop at *LineNo*, *i.e.* the command defines a *breakpoint*. *LineNo* should be a line number in `pram.lis` as discussed in section A.2.4.3.
5. `output` *VarName*
   Prints the value of the specified variable.
6. `output` *ObjectRef.Attribute*
   Prints value of specified attribute.
7. `output` *ObjectRef*`%all`
   Prints all attributes in the object pointed to by *ObjectRef*.
8. `at` *LineNo* `if p = 1 then stop`
   Defines a breakpoint exclusively for processor no 1 (in the current processor set).
9. `at` *LineNo* `if IntTime > 200 then chain; proceed`
   Displays the call chain at *LineNo* when the simulator time has passed 200, and proceeds the execution.
10. `at` *LineNo* `if p = 1 then output %all; proceed`
    Prints the value of all variables in the current block for processor no 1 and proceeds.
11. `trace on` *FirstLine*:*LastLine*
    Makes the debugger print each (SIMULA) source line executed in the range given by the two line numbers.

Note that most of the commands may be abbreviated (ou = output, pro = proceed etc.). Though this debugger is very powerful, nothing can replace the value of fresh air, systematic work, a lot of time, and a cup of coffee.

A.3 Modelling Systolic Arrays and other Synchronous Computing Structures

"I know of only one compute-bound problem that arises naturally in practice for which no systolic solution is known, and I cannot prove that a systolic solution is impossible."

H. T. Kung in *Why Systolic Arrays?* [Kun82].

This section gives some hints on how systolic arrays and similar systems *may* be simulated by help of the CREW PRAM simulator. The text explains how various kinds of systolic arrays may be modelled. It is hoped that the reader will understand how these ideas may be adopted for simulation of other synchronous (globally clocked) computing structures.

A.3.1 Systolic Arrays, Channels, Phases and Stages

The CREW PRAM simulator was originally made for the sole purpose of simulating synchronous CREW PRAM algorithms. However, a few small extensions to the simulator, a PIL "package", and two examples have been made for demonstrating how the simulator can be used for modelling systolic arrays. The adopted solutions are rather ad hoc. More elegant solutions are possible by doing more elaborate redesigns of the simulator.

The file `systool.pil` in the `pil` directory contains PIL code that provides some help in simulating systolic arrays. Its use will be documented by two very simple examples. First it is necessary to explain some central concepts.

Systolic arrays

An important paper on systolic arrays is H. T. Kung's *Why Systolic Architectures?* [Kun82]. It is a comprehensive, compact *and readable* introduction to the topic.

A *systolic array* (or system) is a collection of synchronous processors (often called processing elements or cells) connected in a regular pattern. Information in a systolic system flows between cells in a pipelined fashion. Each cell is designed to perform some specialised operation. When implementing systolic systems in VLSI or similar technologies, it is important to have simple and "repeated" cells, which are interconnected in a regular pattern with mostly short (local) communication paths.

Such modularity and regularity is not a necessity when simulating on the CREW PRAM simulator. Nevertheless, it may be to help in mastering the complexity involved in designing parallel systems.

Channels

Systolic arrays pass data between processing elements on *channels*. These are normally *uni-directional* with one source cell and one destination cell. When simulating a systolic array on the CREW PRAM model, it is natural to let each channel correspond to a specific location in the global memory. This memory area must be allocated before use.[16] Further, each processing element must know the address of every channel it does use during the computation. The channels offered by the simulator may be used in both directions, and they may have multiple readers and multiple writers. If two or more cells write (send) data to the same channel in the same stage, the result will be unpredictable.

[16]This is only true for *integer* channels. When using channels for passing *real* numbers, the current version of the simulator provides a set of preallocated channels.

Three phases in a stage
A systolic system performs a number of computational steps—here called *stages*. One stage consists of every processing element doing the following three phases:

1. *ReadPhase:* Each cell reads data from its input lines (channels).
2. *ComputePhase:* Each cell performs some computation. This is often a simple operation or just a delayed copying (transmitting) of data.
3. *WritePhase:* Each cell writes new data values (computed in the previous phase of this stage) into its output lines (channels).

All cells will finish stage i before they proceed to stage $i+1$. In other words, we have synchronous operation.

These three phases should be considered as *logical phases*. In an implementation it is irrelevant whether phase 3 of stage i is overlapped with phase 1 of stage $i+1$ as long as the data availability rule given below is not violated. Further phase 1 and 2 may be implemented as overlapping or as one phase.

"Data availability rule":
The data written in phase 3 of stage i cannot be read (in phase 1) earlier than in the following stage $i+1$, and later stages.

A.3.2 Example 1: Unit Time Delay

In most (all?) systolic arrays the new values computed from the inputs at the start of stage i are written on the output lines at the end of the same stage—and are available in the next stage. We say that all the cells operate with *unit time delay*, and one time unit is the same as the *duration* of one stage.

The example file `systolicdemo1.pil` models a simple linear array working in this manner. The structure is outlined in Figure A.4. The array does nothing more than copying the received data from left to right in a pipelined fashion. The regularity of this example makes it relatively easy to let n, the number of cells in the linear array, be a parameter to the model. In the simulation program, each kind of cell is described as a procedure, see Figure A.5.

This artificial example has one special processing element for producing input to the array (`InputProducer`) and one for receiving the output from the last cell in the array (`OutputConsumer`). They are not a part of the systolic array, but a convenient way of modelling its environments.

The channels which are used for communication between the cells should be used by the following procedures:

- `SEND(`*ChannelAddressExpression, IntegerExpression*`);`
- *IntegerVariable* `:= RECEIVE(`*ChannelAddressExpression*`);`

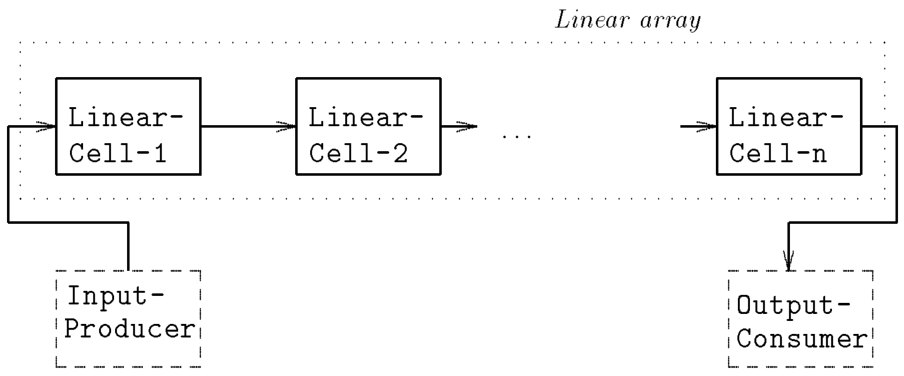


Figure A.4: Linear array simulated in the file `systolicdemo1.pil`.

- *IntegerVariable* `:= RECEIVE_ZEROFY(`*ChannelAddressExpression*`);`
  The value stored at the channel is set to zero after it has been passed to the caller of this procedure.

It is also possible to use `READ` and `WRITE` on the channels (see Section A.2.3.6, but it is *not* recommended.

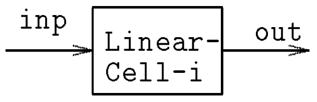
Measuring the length of the phases

As described above, each stage consists of three phases. To achieve proper synchronisation of the array, the user must specify how the operation performed in each stage is distributed on the three phases. This *must be done* by inserting the procedure call `StartComputePhase;` at the end of the read phase, and the call `StartWritePhase;` at the end of the computation phase—see the procedure `LinearCell` in Figure A.5.

This makes it possible for the package `systool.pil` to measure the length of the three phases. At the end of the computation, a call to the procedure `ReportSystolic` will produce a listing of the longest and shortest read, compute and write phase performed, and also the processor number(s) of the processing element(s) which performed those phases. This information may be used to localise bottlenecks in the system. Remember that the processing elements should operate in synchrony. The length of a stage must be the same for all cells—simple or complicated. The structure may be made faster by moving work from the most time consuming cells to under-utilised cells.[17]

In this context, time is measured in number of CREW PRAM simulator time

[17]It is probably the length of the compute phase that is most interesting. Receiving and sending of data are often done in parallel.



```
PROCEDURE LinearCell;
BEGIN
  INTEGER val;
  val := RECEIVE(inp);

  StartComputePhase;
  ! This element has nothing to compute;

  StartWritePhase;
  SEND(out, val);
END_PROCEDURE LinearCell;
```

Figure A.5: Simple cell in a linear systolic array (left), and its description (right).

units, and is specified by the user by calls to the `Use` procedure. The time used by the procedures for accessing the channels (described above) are defined by the constants `t_SEND` etc. at the start of the file `systool.pil`. At the same place you may find constants that specify the maximum lengths that are tolerated for each of the three phases. If your system exceeds one of these limits, an appropriate error message will be given. The limits may be changed by the user.

The modelling of time consumed in the phases may be done at various degrees of detail, and it may be omitted. However, the calls to `StartComputePhase` and `StartWritePhase` may *not* be omitted.

Describing the processing elements (cells)

The procedure `LinearCell` describes the operation of each cell in the linear array which is performed *in each stage*. Therefore, variables declared in the procedure are "new" (initialised to zero) at the start of each stage. Data items that must "live" during the whole computation *must be declared outside the procedures* and before the keyword `BEGIN_PIL_ALG`. These variables are local to each processing element. (See for example the variable `CellNo` in the file `systolicdemo1.pil` which stores the number of each cell in the linear array.)

The variables `inp` and `out` are also of this kind—they store the address of the input channel and output channel of every cell in the linear array. The initialisation of these variables for each processing element corresponds to *"building the hardware"*. This is done by the procedure called `SetUp`, which is called by all processing elements.

`SetUp` may require knowledge of global variables such as the size of the linear array, n. This may be done by passing the value on the *UserStack*, see the calls to `PUSH` in the main program and the first part of the `SetUp` procedure.

General program structure

Figure A.6 outlines the main parts of a general systolic array simulation program using `systool.pil`. It is hoped to be self-explanatory. The reader should note that the `IF` tests in the main loop (`FOR Stage := ...`) must be disjunctive. These tests are used to select what kind of cell a processor should simulate during the stage. One processor may only act as one cell type during a stage.

Figure A.6 outlines only one of several possible ways of structuring a PIL program for simulating systolic arrays. *You should not do changes to the structure shown in Figure A.6 unless you really need to do it*, and only after having studied the details in `systool.pil`.[18]

Please run the example (`systolicdemo1.pil`) and study the output. Important parts of the output are the lines showing the start of each stage, and the contents on the communication channels. The latter is produced by the procedure `TraceChannels`. This procedure should be tailored to the actual structure being simulated. The CREW PRAM simulator time information (`@<<< TIME ...`) is of less value[19] since we have defined a new time unit (stage) at one level higher.

A.3.3 Example 2: Delayed Output and Real Numbers

In the previous example, every cell used exactly one stage to produce new outputs from the inputs. There are cases where this restricts the degree of abstraction that may be used in a model.

Delayed output

Consider the linear array of the previous example. For $n = 5$ this structure delivers an element as output 5 stages after it was received as input. If this is the sole function of what we want to model, it should be possible to represent by a single processing element with delay = 5 stages instead of a linear array consisting of 5 unit delay cells.

This may be achieved by offering an additional parameter *delay* in the `SEND` procedure. The procedure call `SEND(`*addr, val, delay*`)` will do the same as `SEND` described above in Section A.3.2, but the data element will be available on the *channel* at the start of stage $i + delay$ and as long as it is not overwritten (by another `SEND` call) or set to zero by a `RECEIVE_ZEROFY` call.

Once we have this possibility, we may use a single processor to represent a composite structure which makes several computations of varying complexity, and delivers output with differing delays. This possibility may be helpful in top down development of systolic arrays and similar structures.

If you insert

```
#define DELAYS_USED
```

[18]Especially, be very careful when changing the value of the variable `Stage`, and the placement of `StartSystolic`, `EndSystolic` or `StartReadPhase`. Do not forget `StartComputePhase` or `StartWritePhase`, and do not change their order.

[19]This information may be removed by using the filter `removeTIME` on the output produced by the `run` command—just type `run | removeTIME`.

```
#include<systool.pil>

PROCESSOR_SET ProcSetName;
... declaration of variables for the cells

PROCEDURE SetUp;
BEGIN
  ... read eventual parameters from the UserStack
  ... initialize variables and channel-addresses for the
      various cell types
END_PROCEDURE SetUp;

PROCEDURE CellDescription-1; ... one for each cell type
BEGIN
  ... declaration of variables used during one stage
  ... actions in read phase (i.e. calls to RECEIVE)

  StartComputePhase;
  ... actions in compute phase

  StartWritePhase;
  ... actions in write phase (i.e. calls to SEND)
END_PROCEDURE CellDescription;
... more cell descriptions

BEGIN_PIL_ALG
  ... assign processors to ProcSetName;
  ... allocate (integer) channels
  ... PUSH eventual parameters on UserStack;

  FOR_EACH PROCESSOR IN ProcSetName
    SetUp;
    CHECK;
    StartSystolic;

    FOR Stage := 1 STEP 1 UNTIL ... DO
    BEGIN
      StartReadPhase;
      IF p = 1 THEN T_TIO("****** Start of stage no", Stage);

      IF p = ...      THEN CellDescription-1;
      IF p = ...      THEN CellDescription-2;
      ...
    END_FOR;

    EndSystolic; ReportSystolic;
  END_FOR_EACH PROCESSOR;

END_PIL_ALG
```

Figure A.6: General format of program for simulation of systolic arrays using **systool.pil**.

before the place where `systool.pil` is included, you will get access to a version of the `SEND` procedure that requires a *delay* given as parameter.

- `SEND(`*ChannelAddressExpression*, *IntegerExpression*, *DelayExpression*`);`
  DelayExpression must evaluate to an integer ≥ 1. The value 1 corresponds to unit time delay as discussed in the previous example.
- The procedures for receiving data are as before.

It is also possible to use `READ` and `WRITE` on the channels, but it is not recommended.[20]

Using real numbers

The CREW PRAM simulator was originally made for handling integers only. Upon request a simple and ad hoc extension has been made for making it possible to operate on real numbers.

A fixed number of preallocated channels for passing real numbers are provided by the simulator. In the current version 200 channels are provided with addresses 1 to 200. The only difference from the ordinary integer channels is that the real channels cannot be allocated or operated upon by `READ` and `WRITE`.

In the current version, real numbers may only be used together with delays. If you insert

```
#define REALS_USED
```

before the place where `systool.pil` is included, you will get access to the following procedures for passing real numbers between the processing elements.

- `SEND_REAL(`*ChannelAddressExpression*, *RealExpression*, *DelayExpression*`);`
  The delay must be ≥ 1.
- *RealVariable* `:= RECEIVE_REAL(`*ChannelAddressExpression*`);`
- *RealVariable* `:= RECEIVE_REAL_ZEROFY(`*ChannelAddress-Expression*`);`

In addition to:

- `T_TR(`*Text*, *RealExpression*`);`
  Prints the text followed by the value of *RealExpression*.
- `RealArrayPrint(`*ChannelStartAddressExpression, Size*`);`
  Prints the contents of *Size* consecutive real channels starting at and including *ChannelStartAddressExpression*.

[20] `WRITE` corresponds to `SEND` with delay = 1.

Both delayed output, channels passing real numbers, and zerofying are demonstrated on the example file `systolicdemo2.pil`. This artificial example is an extension of the previous example. We have introduced one real channel going down from each of the n cells in the linear array. Further we have introduced a new cell type `RealCell` used in two instances. See the "ASCII-figure" at the start of the file `systolicdemo2.pil`.

`RealCell` is used to illustrate receiving with and without zerofying. `LinearCell` has been extended to compute a real number which is sent downwards.[21] The instances of `LinearCell` demonstrate various delays.

[21] Note how time (`Stage`), state (`val`) and sender (`CellNo`) may be encoded in a number.

Appendix B

Cole's Parallel Merge Sort in PIL

This is a complete listing of Cole's parallel merge sort algorithm implemented in PIL for execution on the CREW PRAM simulator.

B.1 The Main Program

```
% Cole.pil
% ------------------------------------------------------------------
% Cole's Parallel Merge Sort algorithm.
% Complete implementation in PIL on CREW PRAM simulator with detailed
% time modelling.
% ----------------------------------------------------------------------
%    Author: Lasse Natvig  (E-mail: lasse@idt.unit.no)
%    Started: 890822
%    Finished (No known errors): 891004
%    Last changed (improvements): 900524
% Copyright (C) 90-05-30, Lasse Natvig, IDT/NTH, 7034-Trondheim, Norway
% ----------------------------------------------------------------------
% Note that in general, UP is short for Up array, SUP is short for
% SampleUp array, and NUP is short for NewUp array.
%
#include<Sorting>
#include<Cole1.var>
#include<Cole1.mem> ! specifies memory usage ;
#include<Cole1.proc>
#include<Cole2.proc>

BEGIN_PIL_ALG

  ! Loop for testing the algorithm on several problem instances ;
  FOR mm := 2 TO 7 DO
  BEGIN
    nn := power2(mm);
    addr := GenerateSortingInstance(nn, RANDOM);

    ClockReset;
    READ nn FROM UserStackPtr-1;
    UPprocs :=  UPsize(nn);      ! See Cole1.proc ;
    SUPprocs := SUPsize(nn);     ! See Cole1.proc ;
    Use(t_LOAD + t_ADD + t_ADD + t_STORE);
    NoOfProcessors := UPprocs + SUPprocs + SUPprocs;

    Use(2*(t_Malloc+t_STORE) + (t_LOAD+t_SHIFT+t_SUB) + 2*(t_Malloc+t_STORE));
    UPstart := Mem.UserMalloc(NoOfProcessors); ! storage for Arrays ;
    addr    := Mem.UserMalloc(NoOfProcessors); ! storage for ExchangeArea ;
    addr1   := Mem.UserMalloc(2*nn - 1);       ! storage for NodeAddressTable ;
    addr2   := Mem.UserMalloc(2*nn - 1);       ! storage for NodeSizeTable ;

    ! Push variables on the user stack as parameters to the SetUp procedure ;
    Use(6 * t_PUSH);
    PUSH(addr);    PUSH(addr1);    PUSH(addr2);
    PUSH(UPprocs); PUSH(SUPprocs); PUSH(UPstart);

    ASSIGN NoOfProcessors PROCESSORS TO ProcSet1;
    FOR_EACH PROCESSOR IN ProcSet1
      SetUp; ! Read pushed variables and initialize local variables;

      Use(t_FOR);
      FOR Stage := 1 TO LastStage DO
      BEGIN

        ! Since "nothing is done" in Stage 1 and 2.
          If not convinced, study an example ;
        Use(t_IF);
        IF Stage > 2 THEN
          ComputeWhoIsWho;

        Use(t_IF + t_SUB + t_JNEG);
        ! There is no new NUP array to copy in stage 1,2,3 or 5.
          If not convinced, study an example ;
        IF (Stage = 4) OR (Stage > 5) THEN
          CopyNUPtoUP;

        !*** phase 1 of current stage: ;
        ! There is no need to make samples in stage in Stage 1,2 or 4.
          If not convinced, study an example ;
        Use(t_IF + t_SUB + t_JNEG);
        IF (Stage = 3) OR (Stage > 4) THEN
          MakeSamples;

        !*** phase 2 of current stage: ;

        MergeWithHelp;

        ! Store the addresses of the NUP-arrays made in this stage. ;
        ! This is used by CopyNUPtoUP at the beginning of the next stage.;
        StoreArrayAddresses(NUP);

        Use(t_FOR);
      END_FOR;

      IF ProcessorType = NUP AND active AND node = 1 AND index = 1 THEN
        ArrayPrint(ThisAddr, size);

    END_FOR_EACH PROCESSOR;

    T_TITITO("R [Cole.pil] sorted", nn," integers in", ClockRead, " ticks.");
    T_TIO("NoOfProcessors =", NoOfProcessors);

  END_FOR;
END_PIL_ALG
```

B.2 Variables and Memory Usage

B.2.1 Cole1.var

```
% Cole1.var
% ---------

!****** Variables used only by -master- ;
PROCESSOR_SET ProcSet1;
INTEGER mm;
INTEGER nn;
ADDRESS addr, addr1, addr2;

!****** Variables used by all processors ;
INTEGER m;         ! number of levels in complete binary tree ;
INTEGER n;         ! number of integers which is to be sorted (problem size) ;
INTEGER LastStage; ! no of last stage in algorithm ;

INTEGER ProcessorType; ! (= UP, SUP, or NUP) ;
INTEGER CONSTANT UP = 1, SUP = 2, NUP = 3;

INTEGER UPprocs, SUPprocs;! number of UP and SUP (NUP) processors ;
INTEGER NoOfProcessors;
BOOLEAN InsideNode; ! TRUE if this processor is allocated to inside node ;
ADDRESS UPstart;    ! address of the first element of the UP array(s);
ADDRESS ThisAddr;   ! address of element in UP/SUP/NUP permanently allocated ;
                    ! to this processor. ;

ADDRESS ExchangeAreaStart; ! Start of table in global memory used to ;
                           ! exchange locally stored values between processors;

ADDRESS NodeAddressTableStart; ! start of table used to store the ;
                            ! address of the UP, SUP, or NUP array for each ;
                            ! node during the current stage ;
ADDRESS NodeSizeTableStart; ! start of table used to store the ;
                            ! size of the UP, SUP, or NUP array for each ;
                            ! node during the current stage ;
                            ! pr. 890907 only used for SUP array ;


!--- variables which are updated in each stage: ;

INTEGER Stage; ! current stage no in algorithm ( 1..LastStage );
INTEGER HAL;   ! Highest Active Level in current Stage;
INTEGER LAL;   ! Lowest Active Level in current Stage ;
INTEGER ExternalState; ! (=1,2,3), the number of stages this level have been ;
                       ! external including the current stage ;

INTEGER node;  ! no of node (in complete binary tree) which this processor ;
               ! currently is allocated to ;
INTEGER level; ! level containing that node ;
               ! NB node and level may contain wrong values during stages ;
               ! when the processor not is active, i.e. the variable active ;
               ! (see below) is false. ;
INTEGER above; ! the placement of the node (corresponding to this processor) ;
               ! measured in no of levels above the external level ;
INTEGER ProcNoAtLevel; ! no of this processor among the processors of this ;
                       ! kind (UP, SUP or NUP) at this level (1..n);
INTEGER size;  ! size of (UP/SUP/ or NUP) array for nodes of this kind at ;
               ! this level ;

INTEGER index; ! the no of the element served by this processor in the ;
               ! UP/ SUP or NUP array associated with this node. ;
               ! element no in array for this node served by the processor;

BOOLEAN active; ! TRUE if this processor is active in the current stage ;

INTEGER RankInLeftSUP;
INTEGER RankInRightSUP;
```

B.2.2 Cole1.mem

```
% Cole1.mem
% ---------
COMMENT

GLOBAL MEMORY USAGE in Cole1.pil

  location i     : 1'st number  --
  location i+1   : 2'nd number    \
  location i+2   : 3'rd number     ) = n numbers to be sorted
   ...           :    ...         /
  location i+n-1 : n'th number  --
  location i+n   : n                 = problem size, n
  location i+n+1 :    ...       --
  ...            :    ...         \
  ...            :    ...          ) = UP, SUP and NUP arrays,
  ...            :    ...         /    contains NoOfProcessors elements,
  ...            :    ...       --     one for each processor
  ...            :    ...       --
  ...            :    ...         \    ExchangeArea, contains one element
  ...            :    ...          ) = for each processor. (MUST be placed
  ...            :    ...         /    just after the UP,SUP,NUP-arrays)
  ...            :    ...       --
  ...            :    ...       --
  ...            :    ...         )  = NodeAddressTable
  ...            :    ...       --
  ...            :    ...       --
  ...            :    ...         )  = NodeSizeTable
  ...            :    ...       --
                 :    addr      --
                 :    addr1       \
                 :    addr2        ) = parameters to SetUp
                 :    UPprocs     /
                 :    SUPprocs   /
                 :    UPstart   --
  UserStackPtr-> : next free location

end of COMMENT;
```

B.3 Basic Procedures

B.3.1 Cole1.proc

```
% Cole1.proc
% ----------

INTEGER PROCEDURE LowestNodeNo(level); INTEGER level;
BEGIN
  LowestNodeNo := power2(level);
END_PROCEDURE LowestNodeNo;
INTEGER CONSTANT t_LowestNodeNo = t_LOAD + t_power2;
! The time used is modelled at the caller place ;

INTEGER PROCEDURE HighestNodeNo(level); INTEGER level;
BEGIN
  HighestNodeNo := power2(level + 1) - 1;
END_PROCEDURE HighestNodeNo;
INTEGER CONSTANT t_HighestNodeNo = t_LOAD + t_ADD + t_power2 + t_SUB;
! The time used is modelled at the caller place ;

INTEGER PROCEDURE NoOfNodes(level); INTEGER level;
BEGIN
  NoOfNodes := power2(level);
END_PROCEDURE NoOfNodes;
INTEGER CONSTANT t_NoOfNodes = t_LOAD + t_power2;
! The time used is modelled at the caller place ;

PROCEDURE StoreArrayAddresses( type ); INTEGER type;
BEGIN
  Use(t_StoreArray - t_WRITE);
  IF ProcessorType = type AND active AND index = 1 THEN
     WRITE ThisAddr TO NodeAddressTableStart+(node-1)
  ELSE
     Wait(t_WRITE);
END_PROCEDURE StoreArrayAddresses;

PROCEDURE StoreArraySizes( type ); INTEGER type;
BEGIN
  Use(t_StoreArray - t_WRITE);
  IF ProcessorType = type AND active AND index = 1 THEN
     WRITE size TO NodeSizeTableStart+(node-1)
  ELSE
     Wait(t_WRITE);
END_PROCEDURE StoreArraySizes;
INTEGER CONSTANT t_StoreArray = (t_IF + (t_LOAD + t_ADD)*2) +
                                (t_LOAD + t_SUB + t_ADD) + t_WRITE;

PROCEDURE StoreProcessorValue( val );
INTEGER val;
BEGIN
  Use(t_LOAD + t_ADD);
  WRITE val TO ExchangeAreaStart+(p-1);
END_PROCEDURE StoreProcessorValue;
INTEGER CONSTANT t_StoreVal = t_LOAD + t_ADD + t_WRITE;

INTEGER PROCEDURE UPsize(n); INTEGER n;
! Calculates the maximum size (in no of elements) of all the UP arrays ;
! = n + n/2 + n/16 + ... , see Cole88 p 776.                          ;
BEGIN
  INTEGER size;
  Use(t_LOAD + t_STORE + t_SHIFT + t_STORE + t_WHILE);
  size := n;
  n := n//2; ! may be done by shift operation ;
  WHILE n > 0 DO
  BEGIN
    Use(t_LOAD + t_ADD + t_STORE + t_SHIFT + t_STORE + t_WHILE);
    size := size + n;
    n := n//8; ! may be done by shift operation ;
  END_WHILE;
  Use(t_STORE);
  UPsize := size ;
END_PROCEDURE UPsize;
```

```
INTEGER PROCEDURE SUPsize(n); INTEGER n;
! Calculates the maximum size (in no of elements) of all the SUP arrays ;
! = n + n/8 + n/64 + ... , see Cole88 p 776.                            ;
BEGIN
  INTEGER size;
  Use(t_STORE + t_WHILE);
  size := 0;
  WHILE n > 0 DO
  BEGIN
    Use(t_LOAD + t_ADD + t_STORE + t_SHIFT + t_STORE + t_WHILE);
    size := size + n;
    n := n//8; ! may be done by shift operation ;
  END_WHILE;
  Use(t_STORE);
  SUPsize := size ;
END_PROCEDURE SUPsize;
```

PROCEDURE **SetUp**

```
PROCEDURE SetUp;
! Initialization of local processor variables which are unchanged during
  the computation. ;
BEGIN

  PROCEDURE InitProcs;
  ! Initialize processors level by level ;
  BEGIN
    INTEGER ARRAY ProcNo1ofKind(1:3);
    PROCEDURE InitProcKind(type); INTEGER type;
    BEGIN
      INTEGER Initiated;
      INTEGER HowMany;
      INTEGER LevelsAbove;

      Use(t_IF + t_STORE);
      IF type = NUP THEN ! Phase 2 (done by the NUP processors) is not   ;
        LevelsAbove := 1 ! performed at external nodes, thus they are    ;
      ELSE               ! used one level higher up (see Cole88 p 722).  ;
        LevelsAbove := 0;

      Use( (t_LOAD + t_SUB + t_STORE) + (t_LOAD + t_STORE) );
      Initiated := ProcNo1ofKind(type) - 1;
      HowMany := n;

      Use(t_WHILE);
      WHILE HowMany > 0 DO
      BEGIN
        Use(t_IF + t_AND + (t_LOAD + t_ADD + t_IF));
        IF (p > Initiated) AND (p <= Initiated + HowMany) THEN
        BEGIN ! Processor p is of this category ;
          Use( (t_LOAD+t_STORE) + (t_IF+t_STORE) + (t_LOAD+t_SUB+t_STORE));
          above := LevelsAbove;
          InsideNode := (above > 0);
          ProcNoAtLevel := p - Initiated;
        END
        ELSE
          Wait((t_LOAD+t_STORE) + (t_IF+t_STORE) + (t_LOAD+t_SUB+t_STORE));

        Use( (t_LOAD + t_ADD + t_STORE)*2 +
             ( t_IF + t_IF + t_SHIFT + t_STORE*2) );
        Initiated := Initiated + HowMany;
        LevelsAbove := LevelsAbove + 1;
        IF type = UP THEN
          HowMany := IF HowMany = n THEN n//2 ELSE HowMany//8
        ELSE ! SUP or NUP ;
          HowMany := HowMany//8;

        Use(t_WHILE);
      END_WHILE;
```

```
END_PROCEDURE InitProcKind;
```

```
    Use( t_STORE + (t_LOAD + t_ADD + t_STORE)*2);
    ProcNo1ofKind(1) := 1;
    ProcNo1ofKind(2) := UPprocs+1;
    ProcNo1ofKind(3) := UPprocs + SUPprocs + 1;

    InitProcKind(UP);
    InitProcKind(SUP);
    InitProcKind(NUP);
  END_PROCEDURE InitProcs;


  !*** BODY OF SetUp ;

  READ UPstart  FROM UserStackPtr-1;
  READ SUPprocs FROM UserStackPtr-2;
  READ UPprocs  FROM UserStackPtr-3;
  READ NodeSizeTableStart FROM UserStackPtr-4;
  READ NodeAddressTableStart FROM UserStackPtr-5;
  READ ExchangeAreaStart FROM UserStackPtr-6;

  Use(t_LOAD + t_ADD + t_ADD + t_STORE);
  NoOfProcessors := UPprocs + SUPprocs + SUPprocs;

  ! set up the address of the element in UP/SUP/NUP array corresponding ;
  ! to this processor : ;

  Use(t_LOAD + t_ADD + t_STORE);
  ThisAddr := UPstart + (p-1);
  READ n FROM UPstart-1;

  Use( (t_log2 + t_STORE) + (t_MULT + t_STORE) );
  m := log2(n);
  LastStage := 3*m;

  Use( t_IF + (t_LOAD+t_ADD+t_IF+t_STORE) + (t_LOAD+t_ADD+t_IF+t_STORE) );
  ! Time used in longest path modelled ;
  IF p <= UPprocs THEN
    ProcessorType := UP
  ELSE IF p <= UPprocs + SUPprocs THEN
    ProcessorType := SUP;
  IF p > UPprocs + SUPprocs THEN
    ProcessorType := NUP;

  ! read input and place in own array element ;
  Use(t_IF);
  IF ProcessorType = UP THEN
  BEGIN
    INTEGER myNumber;
    Use(t_LOAD + t_SUB);
    READ myNumber FROM ThisAddr-n-1;
    WRITE myNumber TO ThisAddr;
  END
  ELSE
    Wait((t_LOAD + t_SUB) + (t_READ + t_WRITE));

    InitProcs;
END_PROCEDURE SetUp;
```

PROCEDURE **ComputeWhoIsWho**

```
PROCEDURE ComputeWhoIsWho;
BEGIN

  PROCEDURE ComputeSizeOfArrays;
  BEGIN
    INTEGER FirstStageAtThisLevel;

    Use(t_IF + t_IF);
    IF ProcessorType = UP THEN
    BEGIN
      Use(t_IF +
          (t_LOAD + t_SUB + t_MULT + t_ADD + t_STORE) +
          (t_IF + (t_LOAD + t_SUB + t_power2)*2 + t_MULT + t_MIN + t_STORE));
      IF level = m THEN size := 1
      ELSE
      BEGIN
        FirstStageAtThisLevel := (m - level)*2 + 2;
        ! maximum UP-array size at this level is power2(m-level);
        IF Stage >= FirstStageAtThisLevel THEN
          size := MIN(2 * power2(Stage - FirstStageAtThisLevel),
                      power2(m - level))
        ELSE
          size := 0;
      END;
      Wait(t_LOAD + t_AND + t_JNEG);
    END
    ELSE
    IF ProcessorType = SUP THEN
    BEGIN
      Use((t_IF + t_LOAD + t_AND + t_JNEG) +
          (t_LOAD + t_SUB + t_MULT + t_ADD + t_STORE) +
          (t_IF + (t_LOAD + t_SUB + t_power2)*2 + t_MULT + t_MIN + t_STORE));
      IF (level = m) AND (Stage > 2 )  THEN size := 1
      ELSE
      BEGIN
        FirstStageAtThisLevel := (m - level)*2 + 3;
        ! maximum SUP-array size at this level is power2(m-level), i.e. the ;
        ! same as the maximum UP array size. (SUP = UP at last active stage);
        IF Stage >= FirstStageAtThisLevel THEN
          size := MIN(1 * power2(Stage - FirstStageAtThisLevel),
                      power2(m-level))
        ELSE
          size := 0;
      END;
      Wait(0);
    END
    ELSE
    BEGIN ! ProcessorType =  NUP ;
      ! Knows that above >= 1, therefore level must be < m ;
      Use((t_LOAD + t_SUB + t_MULT + t_ADD + t_STORE) +
          (t_IF + (t_LOAD + t_SUB + t_power2)*2 + t_MULT + t_MIN + t_STORE));
      FirstStageAtThisLevel := (m - level)*2 + 1;
      ! maximum NUP-array size at this level is power2(m-level), i.e. the ;
      ! same as the maximum UP array size. (NUP = UP in the next stage)   ;
      IF Stage >= FirstStageAtThisLevel THEN
        size := MIN(2 * power2(Stage - FirstStageAtThisLevel),
                    power2(m-level))
      ELSE
        size := 0;
      Wait(t_IF + t_LOAD + t_AND + t_JNEG);
    END;

  END_PROCEDURE ComputeSizeOfArrays;

  INTEGER NoOfActive; ! No of active processors of this kind at this level ;
```

```
!*** start of body in ComputeWhoIsWho ;

  Use((t_LOAD + t_SUB + t_DIV + t_SUB + t_STORE) +
      (t_LOAD + t_SUB + t_SHIFT + t_SUB + t_STORE) +
      (t_LOAD + t_DIV + t_STORE) + (t_IF + t_STORE));
  LAL := m - ( ( Stage-1)//3 );
  HAL := m - ( ( Stage-1)//2 );
  ExternalState := REM(Stage, 3);
  IF ExternalState = 0 THEN ExternalState := 3;

  Use(t_LOAD + t_SUB + t_STORE);
  level := LAL - above;

  ComputeSizeOfArrays;

  Use(t_IF + t_LOAD + t_NoOfNodes + t_MULT + t_STORE);
  IF level >= 0 THEN
    NoOfActive := NoOfNodes(level) * size
  ELSE
    NoOfActive := 0;

  Use(t_IF +
      (t_LOAD + t_LowestNodeNo + t_LOAD + t_SUB + t_DIV + t_ADD + t_STORE) +
      (t_LOAD + t_SUB + t_DIV + t_MULT + t_SUB + t_STORE) +
      (t_IF + t_STORE));
  IF size > 0 THEN
  BEGIN
    node := LowestNodeNo(level) + (ProcNoAtLevel -1)//size;
    index := ProcNoAtLevel - ( ((ProcNoAtLevel - 1)//size) * size);
    active := (ProcNoAtlevel <= NoOfActive);
  END
  ELSE
  BEGIN
    node := 0; index := 0; active := FALSE;
  END;
END_PROCEDURE ComputeWhoIsWho;
```

PROCEDURE MakeSamples

```
PROCEDURE MakeSamples;
BEGIN
  PROCEDURE Reduce(rate); INTEGER rate;
  BEGIN
    INTEGER UPval, size;
    ADDRESS addr, UPeltAddr;

    Use(t_LOAD + t_SUB);
    READ addr FROM NodeAddressTableStart+(node-1);

    Use((t_LOAD + t_ADD + t_SUB + t_MULT + t_SUB + t_ADD) + t_IF);
    UPeltAddr := addr + size - rate*(size + 1 - index);
    IF UPeltAddr >= addr THEN
    BEGIN
      READ UPval FROM UPeltAddr;
      WRITE UPval TO ThisAddr;
      ! i.e. to corresponding SUP-address ;
    END
    ELSE
      Error("SUP could'nt fetch UPelement");
  END_PROCEDURE Reduce;
  INTEGER CONSTANT t_Reduce = (t_LOAD + t_SUB) + t_R +
    ((t_LOAD + t_ADD + t_SUB + t_MULT + t_SUB + t_ADD) + t_IF) + t_R + t_W;

  INTEGER SampleRate;
  StoreArrayAddresses(UP);

  Use(t_IF + t_LOAD + t_AND);
  IF ProcessorType = SUP AND active THEN
  BEGIN
    Use(t_CONST + t_STORE + t_IF + t_IF + t_STORE);
    SampleRate := 4;
    IF NOT InsideNode THEN
    BEGIN
      IF ExternalState = 2 THEN SampleRate := 2;
      IF ExternalState = 3 THEN SampleRate := 1;
    END;
    Reduce(SampleRate);
  END
  ELSE
    Wait((t_CONST + t_STORE + t_IF + t_IF + t_STORE) + t_Reduce);
END_PROCEDURE MakeSamples;


#include<Order1Merge.proc>
```

PROCEDURE **CopyNUPtoUP**

```
PROCEDURE CopyNUPtoUP;
! Copies the NUP array made in the previous stage to the UP array ;
! for this stage. Performed by the UPprocessors. ;
BEGIN
  ADDRESS addr; ! For Up array elements, this variable represents the
                address of the corresponding NUP array element in the
                previous stage ;
  INTEGER val;
  ! Observation: Copying of NUP to UP should only be done at inside nodes
  except at stage 4, 7, 10.... when it also should be done at external nodes,
  because these nodes were inside nodes in the previous stage ;

  Use(t_IF + t_LOAD + t_AND +
      (t_LOAD + t_SUB + t_DIV + t_JZERO + t_LOAD + t_OR));
  IF ProcessorType = UP AND active AND (REM(Stage-1,3)=0 OR InsideNode) THEN
  BEGIN
    Use(t_LOAD + t_ADD);
    READ addr FROM NodeAddressTableStart+node-1;
    Use(t_LOAD + t_ADD + t_SUB + t_STORE);
    addr := addr+index-1;
    READ val  FROM addr;
    WRITE val TO ThisAddr;
  END
  ELSE
    Wait((t_LOAD + t_ADD)*2 + t_SUB + t_STORE + t_READ + t_READ + t_WRITE);

  !  At the end of the main loop the array addresses are stored for the
  NUP-arrays (in the previous stage). When CopyNUPtoUP is called (at the
  start of this stage) the NodeAddressTable therefore gives
  the mapping from node-no to NUP-processor *as the NUP processors was
  allocated to the NUP-arrays in the previous stage*. ( It is not necces-
  sary to use the NodeSizeTable since the UP-array size for a node in stage i
  is the same as the size of NUP-array in the same node in stage (i-1)).
  Thus the information found in the NodeAddressTable is enough for an UP-
  processor to find its corresponding NUP-processor from the previous stage.

 ! Transfer RankInLeftSUP from NUP to UP processors: ;
  Use(t_IF);
  IF ProcessorType = NUP THEN
    StoreProcessorValue(RankInLeftSUP) ! See Cole2.proc ;
  ELSE
    Wait(t_StoreVal);
  Use(t_IF + t_LOAD + t_SUB + t_ADD); ! assumes that the result of the ;
                  ! subcondition evaluated above was stored locally. ;
  IF ProcessorType = UP AND active AND (REM(Stage-1,3)=0 OR InsideNode) THEN
     READ RankInLeftSUP FROM ExchangeAreaStart+(addr-UPstart)
  ELSE
    Wait(t_READ);

  ! Transfer RankInRightSUP from NUP to UP processors: ;
  Use(t_IF);
  IF ProcessorType = NUP THEN
    StoreProcessorValue(RankInRightSUP)
  ELSE
    Wait(t_StoreVal);

  Use(t_IF + t_LOAD + t_LOAD); !assumes that results were stored locally above;
  IF ProcessorType = UP AND active AND (REM(Stage-1,3)=0 OR InsideNode) THEN
    READ RankInRightSUP FROM ExchangeAreaStart+(addr-UPstart)
  ELSE
    Wait(t_READ);

END_PROCEDURE CopyNUPtoUP;
```

B.3.2 Cole2.proc

```
% Cole2.proc
%
% procedures used in O(1) merging only
%

INTEGER PROCEDURE RateInNextStage;
BEGIN
  INTEGER rate;
  INTEGER ExtState;
  ! InsideNode in this Stage implies InsideNode in next Stage OR ;
  ! first stage as external node in next stage. In both these cases;
  ! the sample rate is 4 ;
  Use(t_IF + (t_LOAD + t_ADD + t_DIV + t_STORE) + 2*(t_IF + t_STORE));
  IF InsideNode THEN
    rate := 4
  ELSE
  BEGIN
    !NOT InsideNode in this Stage implies NOT InsideNode in next stage;
    ExtState := REM(Stage+1,3);
    IF ExtState = 0 THEN rate := 1; ! corresponds to ExtState = 3 ;
    IF ExtState = 2 THEN rate := 2;
  END;
  RateInNextStage := rate;
END_PROCEDURE RateInNextStage;
INTEGER CONSTANT t_RateInNextStage = (t_IF +
             (t_LOAD + t_ADD + t_DIV + t_STORE) + 2*(t_IF + t_STORE));

INTEGER PROCEDURE RateInNextStageBelow;
! Computes the sample rate that will be used on level below this in next stage;
BEGIN
  INTEGER rate;
  INTEGER ExtState;
  Use((t_IF + t_LOAD + t_SUB) + (t_LOAD + t_ADD + t_DIV + t_STORE) +
      2*(t_IF + t_STORE));
  IF (LAL - level) > 1 THEN ! also the level below must be internal ;
    rate := 4
  ELSE
  BEGIN
    ExtState := REM(Stage+1,3);
    IF ExtState = 0 THEN rate := 1; ! corresponds to ExtState = 3 ;
    IF ExtState = 2 THEN rate := 2;
  END;
  RateInNextStageBelow := rate;
END_PROCEDURE RateInNextStageBelow;
INTEGER CONSTANT t_RateInNextStageBelow = (t_IF + t_LOAD + t_SUB) +
    (t_LOAD + t_ADD + t_DIV + t_STORE) + 2*(t_IF + t_STORE);
```

```
INTEGER PROCEDURE Compare1With3(val, val1, val2, val3);
INTEGER val, val1, val2, val3;
BEGIN
  ! NOTE. It is ASSUMED that val1 <= val2 <= val3 ;
  INTEGER pos;
  Use(t_IF + t_IF + t_STORE);
  IF val < val2 THEN
  BEGIN
    IF val < val1 THEN
      pos := 0
    ELSE
      pos := 1;
  END
  ELSE
  BEGIN
    IF val < val3 THEN
      pos := 2
    ELSE
      pos := 3;
  END;
  Compare1With3 := pos;
END_PROCEDURE Compare1With3;
INTEGER CONSTANT t_Compare1With3 = t_IF + t_IF + t_STORE;
```

B.4 Merging in Constant Time

B.4.1 `Order1Merge.proc`

```
% Order1Merge.proc
% ----------------

PROCEDURE MergeWithHelp;
BEGIN

  INTEGER SLrankInParentNUP; ! SL is short for StraddleLeft ;
  INTEGER SRrankInParentNUP; ! SR is short for StraddleRight ;
  ! These must be declared here since they are computed ;
  ! in DoMerge and used in MaintainRanks. ;

  INTEGER CONSTANT t_TryCandidates =
   (t_LOAD + t_SUB + t_ADD + t_STORE + t_IF) +
   (t_READ + ((t_IF +t_SUB + t_AND) + t_LOAD + t_ADD) + t_WRITE) +
   (t_LOAD + t_SUB + t_ADD + t_STORE) +
   (t_READ + (t_IF + t_LOAD + t_ADD) + t_WRITE) +
   ((t_LOAD + t_ADD + t_STORE) + (t_LOAD + t_SUB + t_ADD + t_STORE) + t_IF +
   (t_LOAD + t_ADD) + t_WRITE) +
   (t_WRITE + ((t_LOAD + t_ADD + t_STORE)*2 + t_IF + (t_LOAD + t_ADD)));

  INTEGER CONSTANT t_SubStep1 =  (t_IF) +
    ((t_LOAD + t_SHIFT + t_STORE) + (t_LOAD + t_STORE))*2 + t_ADD +
    2 * (t_StoreVal + (t_IF + t_LOAD + t_SUB + t_ADD) + t_R + t_R + (t_IF) +
    t_R + (t_LOAD + t_ADD) + t_R + t_TryCandidates);

  INTEGER CONSTANT t_find_r_and_t = (t_LOAD + t_SHIFT + t_ADD) + t_R +
    ((t_LOAD + t_ADD + t_SUB + t_STORE) + (t_LOAD + t_ADD)) +
    t_R + (t_LOAD + t_SHIFT + t_ADD) + t_R +
    (t_LOAD + t_ADD + t_IF) + (t_LOAD + t_ADD) + t_R;

  INTEGER CONSTANT t_ss2case1 =  (t_IF + t_StoreVal) +
    (t_IF + t_LOAD + t_SHIFT + t_JZERO) + t_find_r_and_t;

  INTEGER CONSTANT t_ss2case2 =  (t_IF + t_StoreVal) +
    (t_IF + t_LOAD + t_SHIFT + t_JNEG) +  t_find_r_and_t;

  INTEGER CONSTANT t_ReadValues = (t_IF + t_ABS) + (t_LOAD + t_SUB + t_ADD*2) +
    t_R +  (t_IF + t_ADD + t_LOAD) + t_R + (t_IF + t_ADD + t_LOAD) + t_R;
  INTEGER CONSTANT t_SubStep2 = ((t_IF + t_LOAD + t_ADD) +
    t_R + 2* t_StoreArray) +  t_ss2case1 + t_ss2case2 +
    t_IF + t_R + t_StoreVal +
    (t_IF + t_LOAD + t_SHIFT + t_SUB) + t_ReadValues +
    (t_STORE + t_ADD + t_STORE)*2 + t_Compare1With3;

  INTEGER CONSTANT t_ComputeCrossRanks = (t_IF + t_LOAD + t_STORE) +
    (t_IF + t_LOAD + t_SUB + t_JZERO) +  t_SubStep1 + t_SubStep2;

#include<DoMerge.proc>
  INTEGER CONSTANT t_DoMerge = t_DoMergePart1 + t_DoMergePart2;

#include<MaintainRanks.proc>

  !*** BEGIN BODY of MergeWithHelp ;
  ! Main Assumption: Ranks UP(u) -> SUP(v), and UP(u) -> SUP(w) are known.;
  ! these are stored in RankInLeftSUP and RankInRightSUP ;
  BEGIN

    StoreArrayAddresses(SUP);
    StoreArraySizes(SUP);

  ! There is no need to perform DoMerge in Stage 1,2 or 4. If not convinced,
    study an example ;
  ! Assumes that this test is combined with similar test in Cole.pil (mainprog);
  IF (Stage = 3) OR (Stage > 4)  THEN
  BEGIN
    Use(t_IF + t_LOAD + t_JZERO + t_AND + t_AND);
    IF (ProcessorType = UP AND size > 0 AND active AND InsideNode) OR
       (ProcessorType = SUP AND size > 0 AND active AND node > 1) OR
       (ProcessorType = NUP AND size > 0 AND active AND InsideNode) THEN
      DoMerge
    ELSE
      Wait(t_DoMerge);
  END;
END;

! There is no need to perform MaintainRanks in Stage 1,2,3,4 and 6.
  RankInLeft/RightSUP are needed at the first time when the size of UP(u)
  is > 0 and the size of SUP(u.child) > 0. This occur first time in
  Stage 6. (Consider an example) ;
  Use(t_IF + t_SUB + t_JNEG);
  IF (Stage = 5) OR (Stage > 6) THEN
  BEGIN
    MaintainRanks;
  END;

END_PROCEDURE MergeWithHelp;
```

B.4.2 DoMerge.proc

```
% DoMerge.proc

PROCEDURE DoMerge;
BEGIN
  ! NOTE that StoreArraySizes(SUP) and StoreArrayAddresses(SUP) are done ;
  !      just before the call to DoMerge in MergeWithHelp.               ;
  !      This is assumed in SubStep1.                                    ;
  INTEGER RankInParentNUP, RankInThisSUP,  RankInOtherSUP;

  INTEGER SLindex; ! SL = d and SR = f in fig. 2 p 774 in Cole88 ;
  INTEGER SRindex; ! These are the indices of the straddling items in the
                   other SUP array (sibling). They are computed in
                   procedure ComputeCrossRanks. ;
  PROCEDURE ComputeCrossRanks;
  BEGIN
    INTEGER RankInParentUP;
```

PROCEDURE **SubStep1**

```
PROCEDURE SubStep1; ! performed by active, inside UP-processors  ;
BEGIN
  INTEGER r, s, i1, i2;

  PROCEDURE TryCandidates(r, addr);
  INTEGER r; ADDRESS addr;
  BEGIN
    INTEGER CandidateVal;
    ADDRESS SUPelementAddr; ! points to the SUP-array element
     (in global memory) whose RankInParentUP should be set to the
     index (position of own element) of the executing UP-processor.;


     ! The assigned value is passed indirectly by using the
     ExchangeArea. The address of an element in the ExchangeArea
     for a  processor p is given by the address of its element in the
     UP/SUP/NUP array + the value of NoOfProcessors.
     See figure \ref{WorkAreas}. ;

     ! item(r) ;
     Use(t_LOAD + t_SUB + t_ADD + t_STORE + t_IF);
     SUPelementAddr := (addr - 1) + r;

     IF r > 0 THEN
     BEGIN
       READ CandidateVal FROM SUPelementAddr;
       Use(t_IF + t_LOAD + t_ADD);
       IF CandidateVal = i1 THEN
       WRITE index TO SUPelementAddr+NoOfProcessors
       ELSE
         Wait(t_WRITE);
     END
     ELSE
       Wait(t_READ + (t_IF + t_LOAD + t_ADD) + t_WRITE);

     ! Note that r and s may be the same position. In that case we must
      have that s < i2 since r = s, r <= i1 and i1 < i2. Therefore the
      element in position r = s has rank = b in UP(u) <=> it is = i1.
      This case is handled as item(r) above ;

     ! item(s) ;
     Use(t_LOAD + t_SUB + t_ADD + t_STORE);
     SUPelementAddr := (addr - 1) + s;
     READ CandidateVal FROM SUPelementAddr;
     Use((t_IF + t_SUB + t_AND) + t_LOAD + t_ADD);
     IF (CandidateVal < i2) AND (CandidateVal >= i1) THEN
       WRITE index TO SUPelementAddr+NoOfProcessors
     ELSE
       Wait(t_WRITE);

     ! item(r+1) ;
     Use((t_LOAD + t_ADD + t_STORE) +
         (t_LOAD + t_SUB + t_ADD + t_STORE) + t_IF + (t_LOAD + t_ADD));
     r := r + 1;
     SUPelementAddr := (addr - 1) + r;
     IF r < s THEN
       WRITE index TO SUPelementAddr+NoOfProcessors
     ELSE
       Wait(t_WRITE);

     ! item(r+2) ;
     Use((t_LOAD + t_ADD + t_STORE)*2 + t_IF + (t_LOAD + t_ADD));
     r := r + 1;
     SUPelementAddr := SUPelementAddr + 1;
     IF r < s THEN
       WRITE index TO SUPelementAddr+NoOfProcessors
     ELSE
       Wait(t_WRITE);

  END_PROCEDURE TryCandidates;
  ! t_TryCandidates is defined in Order1Merge.proc ;
! *** Substep1 starts here ;
! for each element e in SUP(v) compute its rank in UP(u)   ;

! Find interval [i1,i2> induced by this UP-element. Then find the
items in SUP(v) contained in [i1,i2>. Assign ranks to these elements.
The interval is given by the values i1 and i2, where i1 is the value
stored at ThisAddr for this UP-array processor, and i2 is the value
stored at ThisAddr+1 if not index = size for the same UP-array
processor.  If index = size, ThisAddr is the last element for this
UP-array element, i2 = INFTY, and s = the position of the last
element in SUP(v).
See \ref{Substep1Descr} in co.tex ;

Use(t_IF);
IF ProcessorType = UP THEN
BEGIN
   ! See Figure \ref{Substep1} in co.tex ;
  ADDRESS addr;
  INTEGER LeftChild;

  Use((t_LOAD + t_SHIFT + t_STORE) + (t_LOAD + t_STORE));
  LeftChild := 2*node;
  r := RankInLeftSUP;
  StoreProcessorValue(RankInLeftSUP);

  ! The NodeSizeTable should now contain the size of the SUP-array
  for each node. Similarly, the addresses of the SUP-arrays in the
  NodeAddressTable. See the start of the body of MergeWithHelp ;

  Use(t_IF + t_LOAD + t_SUB + t_ADD);
  IF index < size THEN
    READ s FROM ExchangeAreaStart+(p-1)+1
  ELSE ! Right neighbour processor does not exist, lets point to the
       last element in SUP(v) ;
    READ s FROM NodeSizeTableStart+(LeftChild-1);

  READ i1 FROM ThisAddr;
  Use(t_IF);
  IF index < size THEN
    READ i2 FROM ThisAddr+1
  ELSE
  BEGIN
    Use(t_READ); i2 := INFTY;
  END;
  Use(t_LOAD + t_ADD);
  READ addr FROM NodeAddressTableStart+(LeftChild-1);
  TryCandidates(r, addr);
END of ProcessorType = UP
ELSE
  Wait((t_LOAD + t_SHIFT + t_STORE) + (t_LOAD + t_STORE) + t_StoreVal +
       (t_IF + t_LOAD + t_SUB + t_ADD) + t_R + t_R + (t_IF) +
        t_R + (t_LOAD + t_ADD) + t_R + t_TryCandidates);


! Do the same for each element e in SUP(w). Note that this ;
! computation must be done in two phases (one for SUP(v) and ;
! one for SUP(w)) which not can be done in parallel, since the
! work in both phases are done by the UP(u) processors which ;
! are common for SUP(v) and SUP(w). ;


! for each element e in SUP(w) compute its rank in UP(u)   ;
! Assumes that this test is combined with the one above ;
IF ProcessorType = UP THEN
BEGIN
  ADDRESS addr;
  INTEGER RightChild;

  Use((t_LOAD + t_SHIFT + t_ADD + t_STORE) + (t_LOAD + t_STORE));
  RightChild := 2*node+1;
  r := RankInRightSUP;
  StoreProcessorValue(RankInRightSUP);
```

```
    ! The NodeSizeTable should still contain the size of the    ;
    ! SUP-array for each node. Similarly, the addresses of the  ;
    ! SUP-arrays in the NodeAddressTable.                       ;

    Use(t_IF + t_LOAD + t_SUB + t_ADD);
    IF index < size THEN
      READ s FROM ExchangeAreaStart+(p-1)+1
    ELSE
      READ s FROM NodeSizeTableStart+(RightChild-1);

    READ i1 FROM ThisAddr;
    Use(t_IF);
    IF index < size THEN
      READ i2 FROM ThisAddr+1
    ELSE
    BEGIN
      Use(t_READ); i2 := INFTY;
    END;

    Use(t_LOAD + t_ADD);
    READ addr FROM NodeAddressTableStart+(RightChild-1);
    TryCandidates(r, addr);
  END of ProcessorType = UP
  ELSE
    Wait((t_LOAD+t_SHIFT+t_ADD+t_STORE) + (t_LOAD+t_STORE) + t_StoreVal +
        (t_IF + t_LOAD + t_SUB + t_ADD) + t_R + t_R + (t_IF) +
         t_R + (t_LOAD + t_ADD) + t_R + t_TryCandidates);

END_PROCEDURE SubStep1;
! t_SubStep1 is defined in DoMerge.proc ;
```

PROCEDURE **SubStep2**

```
PROCEDURE SubStep2;
BEGIN
  ! for each element e in SUP(v/w) compute its rank in SUP(w/v) ;
  ! see fig.1. Cole88 p 773, and \ref{Substep2} in co.tex ;

  INTEGER r, t;
  PROCEDURE find_r_and_t;
  BEGIN
    INTEGER dProcNo, ParentUPaddr, ParentSize;

    ! find the processor no which holds d;
    Use(t_LOAD + t_SHIFT + t_ADD);
    READ ParentUPaddr FROM NodeAddressTableStart+(node//2)-1;
    Use((t_LOAD + t_ADD + t_SUB + t_STORE) + (t_LOAD + t_ADD));
    dProcNo := ParentUPaddr + RankInParentUP - UPstart;
    ! RankIn(Left/Right)SUP is now stored in ExchangeArea ;
    READ r FROM ExchangeAreaStart+dProcNo-1;

    Use(t_LOAD + t_SHIFT + t_ADD);
    READ ParentSize FROM NodeSizeTableStart+(node//2)-1;
    Use((t_LOAD + t_ADD + t_IF) + (t_LOAD + t_ADD));
    IF RankInParentUP + 1 > ParentSize THEN
    BEGIN
      t := size;
      Use(t_READ);
    END
    ELSE
      READ t FROM ExchangeAreaStart+dProcNo-1+1;

  END_PROCEDURE find_r_and_t;
  ! t_find_r_and_t is defined in Order1Merge.proc ;


  Use(t_IF + t_LOAD + t_ADD);
  IF ProcessorType = SUP THEN
    READ RankInParentUP FROM ExchangeAreaStart+(p-1)
  ELSE
  Wait(t_READ);

StoreArrayAddresses(UP);
StoreArraySizes(UP);


!*** Case 1 --- e in SUP(v), find rank in SUP(w) ;
! t_ss2case1 starts here ;
Use(t_IF);
IF ProcessorType = UP THEN
  StoreProcessorValue(RankInRightSUP)
ELSE
  Wait(t_StoreVal);

Use(t_IF + t_LOAD + t_SHIFT + t_JZERO);
IF ProcessorType = SUP AND REM(node,2) = 0  THEN
BEGIN
  ! left node, SUP(v) ;
  find_r_and_t;
END
ELSE
  Wait(t_find_r_and_t);
! t_ss2Case1 ends here;

!*** Case 2 --- e in SUP(w), find rank in SUP(v) ;
! t_ss2case2 starts here ;
Use(t_IF);
IF ProcessorType = UP  THEN
  StoreProcessorValue(RankInLeftSUP)
ELSE
  Wait(t_StoreVal);

Use(t_IF + t_LOAD + t_SHIFT + t_JNEG);
IF ProcessorType = SUP AND REM(node,2) > 0 THEN
BEGIN
  ! right node, SUP(w) ;
  find_r_and_t;
END
ELSE
  Wait(t_find_r_and_t);
! t_ss2Case2 ends here;

! The rest of Case 1 and Case 2 is handled in parallel ;
Use(t_IF);
IF ProcessorType = SUP  THEN
BEGIN
  ! Remember that node 1 has no sibling node ;
  INTEGER val, val1, val2, val3, LocalOffset;
  READ val FROM ThisAddr;
  StoreProcessorValue(val);

  BEGIN
    PROCEDURE ReadValues(size);
    INTEGER size;
    BEGIN
      Use((t_IF + t_ABS) + (t_LOAD + t_SUB + t_ADD + t_ADD));
      IF r >= ABS(size) THEN
      ! for the ABS, see the second call to ReadValues below ;
      BEGIN
        ! r+1 corresponds to the value INFTY ;
        Use(t_READ); val1 := INFTY;
      END
      ELSE
        READ val1 FROM ThisAddr-index+size+(r+1);

      Use((t_IF + t_ADD) + t_LOAD); ! Assumes that the long address
                    expression was stored locally (in a register) above ;
      IF r+2 <= t THEN
        READ val2 FROM ThisAddr-index+size+(r+2)
      ELSE
      BEGIN
        val2 := INFTY; Use(t_READ);
      END;
```

```
        Use((t_IF + t_ADD) + t_LOAD);
        IF r+3 <= t THEN
          READ val3 FROM ThisAddr-index+size+(r+3)
        ELSE
        BEGIN
           val3 := INFTY; Use(t_READ);
        END;
      END_PROCEDURE ReadValues;

      Use((t_IF + t_LOAD + t_SHIFT) + t_SUB);
      IF REM(node,2) = 0 THEN
        ReadValues(size) ! this is left child, read from right child ;
      ELSE
        ReadValues(-size); ! this is right child, read from left child ;
    END;
    Use( t_STORE + (t_ADD + t_STORE));
    LocalOffset := Compare1With3(val, val1, val2, val3);
    RankInOtherSUP := r + LocalOffset;

    ! Store the positions in the other SUP of the two straddling items.
    these values are needed in MaintainRanks. ;
    Use( t_STORE + (t_ADD + t_STORE));
    SLindex := RankInOtherSUP;
    SRindex := RankInOtherSUP + 1;
  END
  ELSE
    Wait(t_READ + t_StoreVal + (t_IF + t_LOAD + t_SHIFT + t_SUB) +
         t_ReadValues + ( t_STORE + (t_ADD + t_STORE))*2 +
         t_Compare1With3);

END_PROCEDURE SubStep2;
```

Body of ComputeCrossRanks

```
!*** BEGIN BODY of ComputeCrossRanks ;

Use(t_IF + t_LOAD + t_STORE);
IF ProcessorType = SUP THEN
  RankInThisSUP := index;

! Compute rank in other SUP ;
! see also the processor selection before the call to DoMerge and ;
! ComputeCrossRanks ;

Use(t_IF + t_LOAD + t_SUB + t_JZERO);
IF ProcessorType = SUP AND size = 1 THEN
BEGIN
  ! Boundary case: ;
  ! Since the size of the SUP-array in this node = 1 there is no ;
  ! UP-array in the parent node to help us in computing the rank ;
  ! in the other SUP (as done by the routines SubStep1 and SubStep2) ;
  ! However, since the SUP-arrays are so small it is easy to compute ;
  ! this rank in constant time by using only one comparison. ;
  INTEGER ThisVal, OtherVal;
  ADDRESS OtherAddr;

  Use((t_IF + t_SHIFT) + (t_LOAD + t_ADD));
  IF REM(node, 2) = 0 THEN
    READ OtherAddr FROM NodeAddressTableStart+(node+1)-1
  ELSE
    READ OtherAddr FROM NodeAddressTableStart+(node-1)-1;
  READ OtherVal FROM OtherAddr;

  READ ThisVal FROM ThisAddr;
  Use(t_IF + t_STORE);
  IF ThisVal < OtherVal THEN
    RankInOtherSUP := 0
  ELSE
    RankInOtherSUP := 1;
    Use((t_LOAD + t_STORE) + (t_ADD + t_STORE));
    SLindex := RankInOtherSUP;
    SRindex := RankInOtherSUP + 1;
     ! Note the - sign below ;
     Wait(t_SubStep1 + t_SubStep2 - (
             ( ((t_IF + t_SHIFT) + (t_LOAD + t_ADD)) +
             t_R + t_R + t_R + t_IF + t_STORE) +
             ((t_LOAD + t_STORE) + (t_ADD + t_STORE))));

  END of ProcessorType = SUP and size = 1
  ELSE
  BEGIN
    SubStep1;
    SubStep2;
  END;

END_PROCEDURE ComputeCrossRanks;
```

Body of DoMerge

```
!**************************;
!*** BEGIN body of DoMerge ;
!**************************;
! We want the position of each SUP-element in the NUP-array of its ;
! parent node. This position is computed and stored in the variable ;
! RankInParentNUP.  It is given by the sum of the rank of the element in ;
! SUP(v) and SUP(w). The procedure ComputeCrossRanks computes these ;
! values, and they are stored in RankInThisSUP and RankInOtherSUP upon ;
! return from the procedure. ;

ADDRESS WriteAddress;

! see also the processorselection before the call to DoMerge ;

! t_DoMergePart1 starts here ;
Use(t_IF);
IF ProcessorType = UP OR ProcessorType = SUP THEN
  ComputeCrossRanks
ELSE
  Wait(t_ComputeCrossRanks);

Use(t_IF + t_LOAD + t_ADD + t_STORE);
IF ProcessorType = SUP THEN
  RankInParentNUP := RankInThisSUP + RankInOtherSUP;

! The values SLrankInParentNUP and SRrankInParentNUP are only used
in MaintainRanks which *not* is called before Stage 5, and *not* in
Stage 6. ;
Use(t_IF + (t_LOAD + t_SUB + t_JZERO) + (t_LOAD + t_SUB + t_JNEG));

IF ProcessorType = SUP AND (Stage = 5 OR Stage > 6 ) THEN
BEGIN
  ! Compute SLrankInParentNUP and SRrankInParentNUP:
  Knows SLindex, SRindex, and RankInParentNUP for all SUP-processors.;


  StoreProcessorValue(RankInParentNUP);
  ! Note that the padding used here is not robust, be careful if modifying;
  Use(t_IF);
  IF SLindex < 1 THEN
  BEGIN
    Use(t_STORE);
    SLrankInParentNUP := 0;
    Wait((t_IF + (t_LOAD+t_SHIFT) + (t_LOAD+t_SUB+t_ADD*3+t_STORE) + t_R) -
          t_STORE);
  END
  ELSE
  BEGIN
    INTEGER SLaddr;
    ! Compute processor no (address) corresponding to SLindex.
    NoOfProcessors is added because we want the address
    in the ExchangeArea. ;
    Use(t_IF + (t_LOAD+t_SHIFT) + (t_LOAD+t_SUB+t_ADD*3+t_STORE));
    IF REM(node,2) = 0 THEN ! This is a left child ; !*mark-2;
      SLaddr := (ThisAddr - index + size + SLindex) + NoOfProcessors
    ELSE ! This is a right child ;
      SLaddr := (ThisAddr - index - size + SLindex) + NoOfProcessors;
    READ SLrankInParentNUP FROM SLaddr;
  END;

  Use(t_IF);
  IF SRindex > size THEN
  BEGIN
    Use(t_LOAD + t_SHIFT + t_ADD + t_STORE);
    SRrankInParentNUP := (2*size)+1;
    Wait((t_IF + (t_LOAD+t_SHIFT) + (t_LOAD+t_SUB+t_ADD*3+t_STORE) + t_R) -
         (t_LOAD + t_SHIFT + t_ADD + t_STORE));
  END
  ELSE
  BEGIN
    INTEGER SRaddr;
    Use(t_IF + (t_LOAD+t_SHIFT) + (t_LOAD+t_SUB+t_ADD*3+t_STORE));
    IF REM(node,2) = 0 THEN
      SRaddr := (ThisAddr - index + size + SRindex) + NoOfProcessors
    ELSE
      SRaddr := (ThisAddr - index - size + SRindex) + NoOfProcessors;
    READ SRrankInParentNUP FROM SRaddr;
  END;

END of ProcessorType = SUP AND Stage ...
ELSE
  Wait(t_StoreVal +
       2*( t_IF + (t_IF + (t_LOAD+t_SHIFT) +
                          (t_LOAD+t_SUB+t_ADD*3+t_STORE) + t_R)));


!*** DO THE MERGE ***;
! t_DoMergePart2 starts here ;
StoreArrayAddresses(NUP);

Use(t_IF);
IF ProcessorType = SUP THEN
BEGIN
ADDRESS ParentNUPaddr;
  INTEGER val;
  ! copy the SUP element to its right position in the NUP array
  of its parent node ;
  READ val FROM ThisAddr;

  Use(t_LOAD + t_SHIFT + t_ADD);
  READ ParentNUPaddr FROM NodeAddressTableStart+((node//2)-1);

  Use(t_LOAD + t_ADD + t_SUB + t_STORE);
  WriteAddress := ParentNUPaddr+RankInParentNUP-1;
  ! RankInThisSUP = 1 for the first element in a set. If RankInOtherSUP = 0
  we will have RankInParentNUP = 1 which is the correct position of the
  element because it is in the set. We must subtract 1 to get the correct
  address above as usual because ParentNUPaddr is the address of element
  no 1 and not the address of element no 0 (which do not exist). ;

  WRITE val TO WriteAddress;
END
ELSE
  Wait(t_READ + ((t_LOAD + t_SHIFT + t_ADD) + t_READ) +
       (t_LOAD + t_ADD + t_SUB + t_STORE) + t_WRITE);


! Record the SenderAddress of each NUP-element in the ExchangeArea.
The information is used in MaintainRanks. A negative SenderAddress
indicates that the item is from the left child, a positive address
indicates that it is from the right child. ;

! assumed combined with the if test above ;
IF ProcessorType = SUP THEN
BEGIN
```

```
INTEGER SenderAddress;
```

```
    Use(t_LOAD + t_STORE);
    SenderAddress := index;
    ! Use the address in the ExchangeArea corresponding to the processor
    allocated to the NUP-element just written ;

    Use( (t_IF + t_SHIFT) + (t_LOAD + t_ADD + t_SUB));
    IF REM(node, 2) = 0 THEN ! node is left child ;
       WRITE -SenderAddress TO WriteAddress+NoOfProcessors
    ELSE
       WRITE +SenderAddress TO WriteAddress+NoOfProcessors;
  END of ProcessorType = SUP
  ELSE
    Wait((t_LOAD + t_STORE) + ( t_IF + t_SHIFT ) +
         (t_LOAD + t_ADD + t_SUB) + t_WRITE);

END_PROCEDURE DoMerge;

INTEGER CONSTANT t_DoMergePart1 = t_IF + (t_IF + t_LOAD + t_ADD + t_STORE) +
  (t_IF + (t_LOAD + t_SUB + t_JZERO) + (t_LOAD + t_SUB + t_JNEG)) +
  t_ComputeCrossRanks + (t_StoreVal + 2*(t_IF + (t_IF + (t_LOAD+t_SHIFT) +
                              (t_LOAD+t_SUB+t_ADD*3+t_STORE)+t_R)));

INTEGER CONSTANT t_DoMergePart2 = t_StoreArray + t_IF +
   (t_READ + (t_LOAD + t_SHIFT + t_ADD) + t_READ) +
   (t_LOAD + t_ADD + t_SUB + t_STORE) + t_WRITE +
   ((t_LOAD+t_STORE) + (t_IF+t_SHIFT) + (t_LOAD+t_ADD+t_SUB) + t_WRITE);
```

B.4.3 MaintainRanks.proc

```
% MaintainRanks.proc
% -----------------
  PROCEDURE MaintainRanks;
  BEGIN
    ! Computes the ranks UP(u) -> SUP(v) and UP(u) -> SUP(w) for the *next* ;
    ! stage. This is NUP(u) -> NewSUP(v) and NUP(u) -> NewSUP(w) for *this* ;
    ! stage. These ranks are stored in the variables RankInLeftSUP and      ;
    ! RankInRightSUP in the NUP processors on return from the procedure.    ;
    ! See Cole88 p. 774 ;

    INTEGER SourceProcNo; ! Every NUP-array element corresponds to a ;
    ! SUP-array element, this variable gives the SUP-array processor no ;
    ! The variable is declared here because it is computed in the first ;
    ! of the following two procedures, and used in both of them. ;
```

PROCEDURE **RankInNewSUPfromSenderNode**

```
    PROCEDURE RankInNewSUPfromSenderNode(Sender);
    INTEGER Sender;
    BEGIN
      INTEGER RankInThisNUP;

      ! The two cases that the element e came from SUP(v) or SUP(w) ;
      ! are handled in parallel ;

      ! compute UP(v) -> NUP(v) ;

      Use(t_IF);
      IF ProcessorType = UP THEN
      ! NOTE that active NUP and UP processors at the *same* node occurs at
      stage 6 (when MaintainRanks not is performed) and at stage 8 and all
      later stages. BUT, if NUP(node) does not exist (as active processors)
      any longer, the reason is that UP(node) has reached its maximum size,
      and since NUP(node) = UP(node) there is no need to store more than
      UP(node) (and it is possible to compute UP(v)->NUP(v) even if NUP(v)
      does not exist). ;
      BEGIN
        Use((t_IF + t_LOAD + t_SUB + t_power2) + (t_LOAD + t_ADD + t_STORE));
        IF size = power2(m - level) THEN
        BEGIN
    ! UP(v) = NUP(v) ;
    RankInThisNUP := index;
  END
  ELSE
  BEGIN
    RankInThisNUP := RankInLeftSUP + RankInRightSUP;
  END;
  StoreProcessorValue(RankInThisNUP); !*mark-1;

  ! Store the address of the UP-arrays, needed below ;
  StoreArrayAddresses(UP);

END of ProcessorType = UP
ELSE
  Wait((t_IF + t_LOAD + t_SUB + t_power2) + (t_LOAD + t_ADD + t_STORE) +
        t_StoreVal + t_StoreArray);

! t_Part1 ends / t_Part2 starts ;

! for each element in SUP(v) or SUP(w) find the corresponding ;
! element in UP(v) ;
! What element in UP is this SUP element a copy of? ;

Use(t_IF);
IF ProcessorType = SUP THEN
BEGIN
  ADDRESS UPaddr;
  INTEGER UPeltIndex;
  INTEGER rate;
  INTEGER UParraySize;
  INTEGER RankInThisNewSUP;

  ! compute samplerate which was used in making SUP from UP ;
  Use(t_STORE + t_IF + 2*(t_IF + t_STORE));
  rate := 4;
  IF NOT InsideNode THEN
  BEGIN
    IF ExternalState = 2 THEN rate := 2;
    IF ExternalState = 3 THEN rate := 1;
  END;

  ! find UP-array of same node ;
  Use(t_LOAD + t_ADD);
  READ UPaddr FROM NodeAddressTableStart+(node-1);

  ! find the position in UP-array corresponding to this SUP-element ;
  Use(t_LOAD + t_SUB + t_SHIFT + t_ADD + t_STORE);
  UPeltIndex := 1 + (index - 1)*rate;

  ! The processor no of the UP-array element corresponding to this ;
  ! SUP element is (UPaddr + UPeltIndex - UPstart) ;

  ! The ExchangeArea now contains UP(v) -> NUP(v), see *mark-1 ;
  Use(t_LOAD + t_ADD*2 + t_SUB);
  READ RankInThisNUP FROM ExchangeAreaStart+(UPaddr+UPeltIndex-UPstart)-1;
  ! We now have for each SUP(x) element ;
  ! RankInThisNUP = SUP(x)->NUP(x) ( for x = v or x = w );

  ! NewSUP is made from NUP ("under the names SUP from UP") in the next;
  ! stage, must compute the sample rate that will be used in that stage;

  rate := RateInNextStage;

  ! NewSUP(x) is every rate'th element in NUP(x) ;
  Use(t_IF + (t_LOAD + t_SUB + t_SHIFT + t_ADD + t_STORE));
  IF rate > 0 THEN
    RankInThisNewSUP := 1 + (RankInThisNUP - 1)//rate;
  ! this is SUP(x) -> NewSUP(x), save this value and the address ;
  ! of the SUP-arrays to make it possible for the NUP processors ;
  ! to get the correct value below . ;
  StoreProcessorValue(RankInThisNewSUP);
  StoreArrayAddresses(SUP);

END of ProcessorType = SUP
```

```
    ELSE
      Wait( (t_STORE + t_IF + 2*(t_IF + t_STORE)) + (t_LOAD + t_ADD) + t_R +
            (t_LOAD + t_SUB + t_SHIFT + t_ADD + t_STORE) +
            (t_LOAD + t_ADD*2 + t_SUB) + t_R + t_RateInNextStage +
            (t_IF + (t_LOAD + t_SUB + t_SHIFT + t_ADD + t_STORE)) +
            t_StoreVal + t_StoreArray);
    ! t_Part2 - ends here ;

    ! Let NUP(u) processors get their value of the rank ;
    ! NUP(u) -> NewSUP(v/w) if their NUP-elt came from SUP(v/w) ;

    Use(t_IF);
    IF ProcessorType = NUP THEN
    BEGIN
      ADDRESS SUPxAddr;
      Use(t_IF);
      IF Sender < 0 THEN
      BEGIN ! from left child ;
        Use(t_LOAD + t_SHIFT + t_ADD);
        READ SUPxAddr FROM NodeAddressTableStart+(2*node)-1;
        Use((t_LOAD + t_SUB*2 + t_STORE) + t_ADD);
        SourceProcNo := SUPxAddr + (-Sender) - UPstart;
        READ RankInLeftSUP FROM ExchangeAreaStart+(SourceProcNo-1);
      END
      ELSE
      BEGIN
        IF Sender = 0 THEN Error(" sender = 0");
        ! from right child ;
        Use(t_LOAD + t_SHIFT + t_ADD);
        READ SUPxAddr FROM NodeAddressTableStart+(2*node+1)-1;
        Use((t_LOAD + t_SUB*2 + t_STORE) + t_ADD);
        SourceProcNo := SUPxAddr + (+Sender) - UPstart;
        READ RankInRightSUP FROM ExchangeAreaStart+(SourceProcNo-1);
      END;
    END
    ELSE
      Wait(t_IF + (t_LOAD + t_SHIFT + t_ADD) + t_READ +
          ((t_LOAD + t_SUB*2 + t_STORE) + t_ADD) + t_READ);

    ! Remember that it were the values for the *next* stage of
      RankInLeft/RightSUP which were computed above. ;

  END_PROCEDURE RankInNewSUPfromSenderNode;

  INTEGER CONSTANT t_Part1 = t_IF +
    (t_IF + t_LOAD + t_SUB + t_power2) + (t_LOAD + t_ADD + t_STORE) +
    t_StoreVal + t_StoreArray;

  INTEGER CONSTANT t_Part2 =
    t_IF + (t_STORE + t_IF + 2*(t_IF + t_STORE)) + (t_LOAD + t_ADD) + t_R +
    (t_LOAD + t_SUB + t_SHIFT + t_ADD + t_STORE) +
    (t_LOAD + t_ADD*2 + t_SUB) +  t_R + t_RateInNextStage +
    (t_IF+ (t_LOAD + t_SUB + t_SHIFT + t_ADD + t_STORE) +
    t_StoreVal + t_StoreArray);

  INTEGER CONSTANT t_RankInNewSUPfromSenderNode =  t_Part1 + t_Part2 +
    (t_IF + (t_IF + (t_LOAD + t_SHIFT + t_ADD) + t_READ +
     ((t_LOAD + t_SUB*2 + t_STORE) + t_ADD) + t_READ));
```

PROCEDURE **RankInNewSUPfromOtherNode**

```
  PROCEDURE RankInNewSUPfromOtherNode(Sender);
  INTEGER Sender;
  BEGIN
    INTEGER r, t ;
    INTEGER LocalOffset;

    PROCEDURE ComputeLocalOffset(OtherNUPaddr, rate);
    ADDRESS OtherNUPaddr;
    INTEGER rate;
    BEGIN
      INTEGER OwnVal, val1, val2, val3;
  ! Get the maximum 3 elements in positions r+1, t-1 and t ;
  ! See fig 2. Cole88 p 774 ;
  ! This procedure is similar to ReadValues in DoMerge, but the address
  calculation is different because we must use 'rate' to obtain the
  correct mapping between NewSUP(x) and NUP(x), see *mark-7 below ;

  ! Item x is read from position OtherNUPaddr + [(x-1)*rate] ;
  Use(t_LOAD + t_SHIFT + t_ADD);
  READ val1 FROM OtherNUPaddr+(r*rate);

  Use((t_ADD + t_IF) + (t_LOAD + t_SHIFT + t_ADD));
  IF r+2 <= t THEN
  BEGIN
    READ val2 FROM OtherNUPaddr+((r+1)*rate);
  END
  ELSE
  BEGIN
    Use(t_READ); val2 := INFTY;
  END;

  Use((t_ADD + t_IF) + (t_LOAD + t_SHIFT + t_ADD));

  IF r+3 <= t THEN
  BEGIN
    READ val3 FROM OtherNUPaddr+((r+2)*rate);
  END
  ELSE
  BEGIN
    Use(t_READ); val3 := INFTY;
  END;

  READ OwnVal FROM ThisAddr;
  LocalOffset := Compare1With3(OwnVal, val1, val2, val3);

END_PROCEDURE ComputeLocalOffset;

INTEGER dRank, fRank;
INTEGER NUPvAddr, NUPwAddr;
INTEGER NUPvANDwSize;
! The elements d and f are the two elements from the other SUP-array
which straddle e (e = this element). See fig 2 in Cole88 p 774.
dRank is the rank of the element d in NUP(u), similarly for fRank.;
! These values are computed in the procedure DoMerge. In that
procedure they are stored in the variables SLrankInParentNUP and
SRrankInParentNUP ;


! Pass first dRank, then fRank, from the SUP to the NUP-processors: ;
! SourceProcNo was calculated above in RankInNewSUPfromSenderNode;


! t_part2_1 starts here ;
Use(t_IF);
IF ProcessorType = SUP THEN
  StoreProcessorValue(SLrankInParentNUP)
ELSE
  Wait(t_StoreVal);

Use(t_IF + t_LOAD + t_ADD);
IF ProcessorType = NUP THEN
  READ dRank FROM ExchangeAreaStart+(SourceProcNo-1)
ELSE
  Wait(t_READ);

Use(t_IF);
IF ProcessorType = SUP THEN
  StoreProcessorValue(SRrankInParentNUP)
ELSE
  Wait(t_StoreVal);

Use(t_IF + t_LOAD + t_ADD);
IF ProcessorType = NUP THEN
  READ fRank FROM ExchangeAreaStart+(SourceProcNo-1)
ELSE
  Wait(t_READ);
```

```
! This code is needed because NewSUP(x) does not exist. Instead we
find a given NewSUP(x) item in NUP(x) by considering the sample
rate that will be used in the next stage one level below to make
NewSUP(x) from NUP(x). ;
! In the code below ( see *mark-4 and *mark-5 ) every NUP-processor
must know the address of the NUP-arrays (if it exist) of its two
children. If they do not exist, the address of the UP-arrays of the
children must be known.
If there are active NUP processors in node x, there are active NUP
processors in the left and right child of x if the size of the NUP-
array at node x is smaller than (1/4)*MaxNUPsize(x.level).;

! Assumes that size and address of NUP procs are stored
in the Size- and Address Table (see *mark-6 below);

! t_part2_2 starts here ;
Use(t_IF + t_LOAD + t_SUB + t_power2 + t_SHIFT + t_SUB + t_JNEG);
IF ProcessorType = NUP AND size < power2(m - level)//4 THEN
BEGIN
  ! NUP-array should exist in right and left child ;
  Use(t_IF + t_LOAD + t_SHIFT + t_ADD );
  IF Sender > 0 THEN
    READ NUPvAddr FROM NodeAddressTableStart+(node*2)-1
  ELSE
    READ NUPwAddr FROM NodeAddressTableStart+((node*2)+1)-1;
  Use(t_LOAD); ! Assumes use of offset computed above ;
  READ NUPvANDwSize FROM NodeSizeTableStart+(node*2)-1;
END
ELSE
  Wait((t_IF + t_LOAD + t_SHIFT + t_ADD) + t_R + t_LOAD + t_R);

StoreArrayAddresses(UP);
StoreArraySizes(UP);
Use(t_IF); ! Assumed combined with the IF statement above ;
IF ProcessorType = NUP AND size >= power2(m - level)//4 THEN
BEGIN
  ! UP-array should exist in right and left child ;
  ! UP-array is used as NUP array ;
  Use(t_IF + t_LOAD + t_SHIFT + t_ADD);
  IF Sender > 0 THEN
    READ NUPvAddr FROM NodeAddressTableStart+(node*2)-1
  ELSE
    READ NUPwAddr FROM NodeAddressTableStart+((node*2)+1)-1;
  Use(t_LOAD);
  READ NUPvANDwSize FROM NodeSizeTableStart+(node*2)-1;
END
ELSE
  Wait((t_IF + t_LOAD + t_SHIFT + t_ADD) + t_R + t_LOAD + t_R);

! t_part2_3 starts here ;
Use(t_IF);
IF ProcessorType = NUP THEN
BEGIN
  INTEGER rate;

  PROCEDURE find_r_and_t;
  BEGIN
    ! Knows that d is to the left of e, the position of e is index
    and that f is to the right of e. d, e, and f are all inside the
    same NUP(x) array.
    The address of the 1st element (with index=1) in this array
    for this node is "ExchangeAreaStart + p - index". Therefore
    the address of element no <pos> (numbered 1..size) is "<pos> - 1"
    added to this value. ;
    Use(t_IF + t_LOAD + t_ADD + t_SUB + t_ADD);
    IF dRank > 0 THEN
      READ r FROM (ExchangeAreaStart+p-index)+(dRank-1)
    ELSE
    BEGIN
      Use(t_READ);
      r := 0;
    END;
  Use(t_IF + t_LOAD + t_SUB + t_ADD); ! Assumed combined with above;
  IF fRank <= size ! size of own NUP-array ; THEN
    READ t FROM (ExchangeAreaStart+p-index)+(fRank-1)
  ELSE
  BEGIN
    t := NUPvANDwSize//rate;
    Wait(t_READ); ! Assumed not simpler than the THEN-clause ;
  END;
END_PROCEDURE find_r_and_t;


! r and t are the ranks of d and f in the NewSUP for the same
node as d and f came from (SUP). These values have just been
computed in procedure RankInNewSUPfromSenderNode;


! The rate used in the next stage have been computed before
by the SUP-processors in procedure RankInNewSUPfromSenderNode.
However, in this case we need the rate that will be used in the
next stage in the CHILD nodes of this node. ;

Use(t_STORE);
rate := RateInNextStageBelow;

StoreProcessorValue(RankInLeftSUP);
Use(t_IF);
IF Sender > 0 THEN
BEGIN
  ! This is from right child SUP(w), d and f from the left. ;
  find_r_and_t;
END
ELSE
  Wait(t_find);

StoreProcessorValue(RankInRightSUP);
Use(t_IF);
IF Sender < 0 THEN
BEGIN
  ! This is from left child SUP(v), d and f from the right ;
  find_r_and_t;
END
ELSE
  Wait(t_find);

Use(t_IF);
IF Sender > 0 THEN
BEGIN ! *mark-7 ;
  ! r and t points to elements in NewSUP(v). (This is the case
  shown in fig. 2 in Cole88). Note that NewSUP does *not* exist yet.
  However we know that NewSUP(v) will be made by the procedure
  MakeSamples from NUP(v) in the next stage. Thus the positions r and t
  in NewSUP(v) implicitly gives the positions in NUP(v) for the
  elements which must be compared with e.
  If the samplerate used to make NewSUP from NUP is rate, the position
  of the element corresponding to r in NUP(v) is given by 1+(r-1)*rate;

  ! NUPvAddr was read above, *mark-4;
  ComputeLocalOffset(NUPvAddr, rate);

END of Sender > 0
ELSE
BEGIN
  IF Sender < 0 THEN
  BEGIN
    ! r and t points to elements in NewSUP(w). ;
    ! see comments for the case Sender > 0 above ;

    ! NUPwAddr was read above, *mark-5;
    ComputeLocalOffset(NUPwAddr, rate);
  END
  ELSE
    Error(" Should not occur");
END;
```

```
  END of ProcessorType = NUP
  ELSE
    Wait(t_STORE + t_RateInNextStageBelow +
         (t_StoreVal + t_IF + t_find)*2 + (t_IF + t_ComputeLocalOffset));

  ! Can be appended to IF-test above ;
  IF ProcessorType = NUP THEN
  BEGIN
    Use(t_IF + t_LOAD + t_ADD + t_STORE);
    IF Sender < 0 THEN
    BEGIN
      ! This item is from left, other child is right ;
      RankInRightSUP := r + LocalOffset;
    END
    ELSE
    BEGIN
      ! This item is from right, other child is left ;
      RankInLeftSUP := r + LocalOffset;
    END;
  END
  ELSE
    Wait(t_IF + t_LOAD + t_ADD + t_STORE);
END_PROCEDURE RankInNewSUPfromOtherNode;

INTEGER CONSTANT t_part2_1 = 2*(t_IF + t_StoreVal + (t_IF+t_LOAD+t_ADD) +
                             t_READ);
INTEGER CONSTANT t_part2_2 =
  (t_IF + t_LOAD + t_SUB + t_power2 + t_SHIFT + t_SUB + t_JNEG) +
  (t_IF + t_LOAD + t_SHIFT + t_ADD  + t_R + t_LOAD + t_R) +
  2*t_StoreArray + t_IF +
  ((t_IF + t_LOAD + t_SHIFT + t_ADD) + t_R + t_LOAD + t_R);

INTEGER CONSTANT t_ComputeLocalOffset = (t_LOAD + t_SHIFT + t_ADD) +
  t_R + ((t_ADD + t_IF) + (t_LOAD + t_SHIFT + t_ADD)) +
  t_R + ((t_ADD + t_IF) + (t_LOAD + t_SHIFT + t_ADD)) +
  t_R + t_R + t_Compare1With3;

INTEGER CONSTANT t_find = (t_IF + t_LOAD + t_ADD + t_SUB + t_ADD) +
  t_READ + (t_IF + t_LOAD + t_SUB + t_ADD) + t_READ;

INTEGER CONSTANT t_part2_3 = t_IF +
  t_STORE + t_RateInNextStageBelow + (t_StoreVal + t_IF + t_find)*2 +
  (t_IF + t_ComputeLocalOffset) +
  (t_IF + t_LOAD + t_ADD + t_STORE);

INTEGER CONSTANT t_RankInNewSUPfromOtherNode = t_part2_1 + t_part2_2 +
                                               t_part2_3;
```

Body of MaintainRanks

```
  !****** BODY OF MaintainRanks ;
  ! It is only necessary to perform MaintainRanks for nodes with NUP-array
  which is smaller than the maximum size of NUP-arrays at the same level,
  i.e. MergeWithHelp must be performed also in the next stage. This maximum
  size is given by power2(m - level)    ;

  INTEGER SenderAddr;


  Use((t_IF + t_AND) + (t_LOAD + t_SUB + t_power2 + t_SUB + t_JNEG));
  IF ProcessorType = NUP AND active AND size < power2(m - level) THEN
  BEGIN
    ! read sender address of this element ;
    Use(t_LOAD + t_ADD);
    READ SenderAddr FROM ExchangeAreaStart+(p-1);
  END
  ELSE
  BEGIN
    SenderAddr := 0; ! this is done for robustness ;
    Wait(t_LOAD + t_ADD + t_READ);
  END;

  Use(t_IF + t_AND + t_OR + t_LOAD + t_AND + t_OR);
  ! Assumed combined with the test above and simplified ;
  IF (ProcessorType = UP  AND active) OR
     (ProcessorType = SUP AND active AND size < power2(m - level)) OR
     (ProcessorType = NUP AND active AND size < power2(m - level)) THEN
  BEGIN
    RankInNewSUPfromSenderNode(SenderAddr);
  END
  ELSE
    Wait(t_RankInNewSUPfromSenderNode);



  StoreArrayAddresses(NUP);     !*mark-6;
  StoreArraySizes(NUP);

  Use(t_IF); ! Assumes that the result of the same test was stored above;
  IF (ProcessorType = UP  AND active) OR
     (ProcessorType = SUP AND active AND size < power2(m - level)) OR
     (ProcessorType = NUP AND active AND size < power2(m - level)) THEN
  BEGIN
    RankInNewSUPfromOtherNode(SenderAddr);
  END of processor-selection
  ELSE
    Wait(t_RankInNewSUPfromOtherNode);

  ! The ranks NUP(u) -> NewSUP(v) are stored in RankInLeftSUP for the
  NUP processors and similarly in the RankInRightSUP for NUP(u) -> NewSUP(w)

  The addresses of the NUP-arrays are stored at end of this stage (see
  the main loop), and this info is used by the UP-processors to read (copy)
  these Ranks in the procedure CopyNUPtoUP at the start of the next stage.;

END_PROCEDURE MaintainRanks;
```

Appendix C

Batcher's Bitonic Sorting in PIL

This is a complete listing of Batcher's bitonic sorting algorithm [Bat68, Akl85] implemented in PIL for execution on the CREW PRAM simulator.

C.1 The Main Program

```
% Bito.pil
% -------------------------------------------------------------------
% Bitonic sorting with detailed time modelling.
% A bitonic sorting network is emulated by using n/2 processors to act
% as the active column of comparators in each pass during each stage.
% ----------------------------------------------------------------------
%    Author: Lasse Natvig  (E-mail: lasse@idt.unit.no)
%    Last changed (improvements): 900601
% Copyright (C) 90-06-01, Lasse Natvig, IDT/NTH, 7034-Trondheim, Norway
% ----------------------------------------------------------------------

#include<Sorting>

PROCESSOR_SET ComparatorColumn;
INTEGER n, m;
ADDRESS addr;

BEGIN_PIL_ALG

  FOR m := 2 TO 7 DO
  BEGIN
    n := power2(m);
    addr := GenerateSortingInstance(n, RANDOM);

    ClockReset;
    Use(2*t_PUSH);
    PUSH(addr);
    PUSH(n);

    ASSIGN n//2 PROCESSORS TO ComparatorColumn;
```

```
FOR_EACH PROCESSOR IN ComparatorColumn
BEGIN
  INTEGER n, m, Stage, Pass;
  ADDRESS Addr, UpInAddr, LowInAddr;
  INTEGER UpInVal, LowInVal;
  INTEGER GlobalNo ; ! Global Comparator No, (0,1,2,3, ...) ;
  INTEGER sl; ! sequence length is always a power of two ;

  PROCEDURE ComputeInAddresses;
  BEGIN
    INTEGER sn; ! sequence number, (0,1,2,3...);
    ADDRESS sa; ! sequence address ;
    INTEGER LocalNo; ! Local Comparator No inside this sequence (0,1,2,..);

    Use(t_IF + MAX( (t_LOAD + t_power2 + t_STORE),
                    (t_IF + t_LOAD + t_SHIFT + t_STORE)));
    IF Pass = 1 THEN
      sl := power2(Stage)
    ELSE IF Pass > 2 THEN
      sl := sl//2;

    Use( (t_LOAD + t_SHIFT + t_SHIFT + t_STORE) +
         (t_LOAD + t_SHIFT + t_ADD + t_STORE) +
         (t_LOAD + t_SHIFT + t_MOD + t_STORE));
    sn := (GlobalNo*2)//sl;
    sa := Addr + sn*sl;
    LocalNo := MOD(GlobalNo, (sl//2));


    Use(t_IF + MAX(((t_LOAD + t_ADD + t_STORE) +
                    (t_LOAD + t_SHIFT + t_ADD + t_STORE)),
                    (2*(t_LOAD + t_SHIFT) + t_SUB + t_JNEG + t_STORE) +
                    (t_IF + MAX( (2*(t_LOAD+t_SHIFT+t_ADD+t_STORE)),
                                 ((t_LOAD+t_SHIFT+t_ADD+t_ADD+t_STORE) +
                                  (t_LOAD+t_SHIFT+t_SUB+t_STORE))))));

    ! This way of modelling time used gives easy analysis because every call
      to the procedure ComputeInAddresses takes the same amount of time.
      Since the variable Pass always will have the same value for all processors,
      the case Pass = 1 occur for all processors simultaneously. Therefore, a
      faster execution might be obtained by placing appropriate Use statements
      in the THEN and ELSE part of IF Pass = 1 ...
      However, for the test IF UpperPart ... time modelling with
      Use( t_IF + MAX(Then-Clause, Else-clause)) is most appropriate, since we
      simultaneously have processors in both branches --- and must wait for
      the slowest branch to finish. ;
    IF Pass = 1 THEN
    BEGIN
      UpInAddr := sa + LocalNo;
      LowInAddr := UpInAddr + (sl//2);
    END
    ELSE
    BEGIN
      BOOLEAN UpperPart;
      UpperPart := (LocalNo*2) < (sl//2);
      IF UpperPart THEN
      BEGIN
        UpInAddr := sa + (LocalNo*2);
        LowInAddr := UpInAddr + sl//2;
      END
      ELSE
      BEGIN
        LowInAddr := sa + (LocalNo*2) + 1;
        UpInAddr := LowInAddr - sl//2;
      END;
    END;
  END_PROCEDURE ComputeInAddresses;
```

```
PROCEDURE Comparator;
BEGIN
  PROCEDURE Swap(a,b);
  NAME a, b; INTEGER a, b;
  BEGIN
    INTEGER Temp;
    Temp := a; a := b; b := Temp;
  END_PROCEDURE Swap;
  INTEGER CONSTANT t_Swap = 3*(t_LOAD + t_STORE);

  INTEGER Type; ! See Fig. 4.6 Selim G Akl's book on parallel sorting;
  BEGIN ! Compute type ;
    INTEGER Interval;

    Use( (t_LOAD + t_SUB + t_power2 + t_STORE) +
         (t_IF + t_LOAD + t_SHIFT + t_STORE));
    Interval := power2(Stage - 1); ! this is from Akl85) book;
    IF IsEven(GlobalNo//Interval) THEN
      Type := 0
    ELSE
      Type := 1;
  END;

  Use(t_IF + (t_IF + t_Swap));
  IF Type = 0 THEN
  BEGIN
     IF LowInVal < UpInVal THEN
       Swap(LowInVal, UpInVal);
  END
  ELSE
  BEGIN
      IF LowInVal > UpInVal THEN
        Swap(LowInVal, UpInVal);
  END;

  Use(t_LOAD + t_SHIFT + t_ADD);
  WRITE UpInVal TO  Addr+(GlobalNo*2);
  WRITE LowInVal TO Addr+(GlobalNo*2)+1;
END_PROCEDURE Comparator;
```

```
      !************* MAIN PROGRAM *****************;
      ! SetUp: ;

      Use(t_LOAD);
      READ n FROM UserStackPtr-1;
      Use(t_SUB);
      READ Addr FROM UserStackPtr-2;
      Use((t_LOAD + t_log2 + t_STORE) + (t_LOAD + t_SUB + t_STORE));
      m := log2(n);
      GlobalNo := p - 1;

      !*** MAIN LOOP:   ;

      Use(t_STORE !--- set to 1 ---; + t_IF);

      FOR Stage := 1 TO m DO
      BEGIN

        Use(t_STORE + t_IF);
        FOR Pass := 1 TO Stage DO
        BEGIN
          ComputeInAddresses;
          READ UpInVal FROM UpInAddr;
          READ LowInVal FROM LowInAddr;

          Comparator;
          Use(t_LOAD + t_ADD + t_STORE + t_IF);

        END_FOR;

        Use(t_LOAD + t_ADD + t_STORE + t_IF);
      END_FOR;
    END;
    END_FOR_EACH PROCESSOR;

    T_TITIO("P bitonic", n, " ", ClockRead);
    T_TI("P bitonicCost", n); OUTTEXT(" ");
    OUTINT((n//2)*ClockRead, 10); OUTIMAGE;
    T_TO(" ");

    FreeProcs(ComparatorColumn);
    BEGIN
      INTEGER dummy; ! to avoid warning ;
      dummy := POP;
      dummy := POP;
    END;

  END_FOR;
END_PIL_ALG
```

Bibliography

[AAN84] Kjell Øystein Arisland, Anne Cathrine Aasbø, and Are Nundal. VLSI parallel shift sort algorithm and design. *INTEGRATION, the VLSI journal*, pages 331–347, 1984.

[AC84] Loyce M. Adams and Thomas W. Crockett. Modeling Algorithm Execution Time on Processor Arrays. *IEEE Computer*, pages 38–43, July 1984.

[ACG86] Sudhir Ahuja, Nicholas Carriero, and David Gelernter. Linda and Friends. *IEEE Computer*, 19(8):26–34, August 1986.

[AG89] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing.* The Benjamin/Cummings Publishing Company, Inc., 1989.

[Agg89] Alok Aggarwal. A Critique of the PRAM Model of Computation. In *Proceedings of the NSF - ARC Workshop on Opportunities and Constraints of Parallel Computing ([San89a])*, pages 1–3, 1989.

[AHMP86] H. Alt, T. Hagerup, K. Mehlhorn, and F. P. Preparata. Deterministic Simulation of Idealized Parallel Computers on More Realistic Ones. In *Mathematical Foundations of Computer Science 1986, Bratislava, Czechoslovakia, Lecture Notes in Computer Science no 233.*, pages 199–208, 1986.

[AHU74] A. Aho, J. Hopchroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1974.

[AHU82] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1982.

[Akl85] Selim G. Akl. *Parallel Sorting Algorithms.* Academic Press, Inc., Orlando, Florida, 1985.

[Akl89] Selim G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice Hall International, Inc., Englewood Cliffs, New Jersey, 1989.

[AKS83] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. *Combinatorica*, 3(1):1–19, 1983.

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computer capabilities. In *AFIPS joint computer conference Proceedings*, volume 30, pages 483–485, 1967.

[And86] Richard Anderson. *The Complexity of Parallel Algorithms.* PhD thesis, Stanford, 1986.

[Bas87] A. Basu. Parallel processing systems: a nomenclature based on their characteristics. *IEE Proceedings*, 134(3):143–147, May 1987.

[Bat68] K. E. Batcher. Sorting networks and their applications. In *Proc. AFIPS Spring Joint Computer Conference*, pages 307–314, 1968.

[BCK90] R. Bagrodia, K. M. Chandy, and E. Kwan. UC: A Language for the Connection Machine. In *Proceedings of SUPERCOMPUTING'90, New York, November 12–16*, pages 525–533, 1990.

[BDHM84] Dina Bitton, David J. DeWitt, David K. Hsiao, and Jaishankar Menon. A Taxonomy of Parallel Sorting. *Computing Surveys*, 16(3):287–318, September 1984.

[BDMN79] Graham M. Birtwistle, Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. *SIMULA BEGIN, second edition.* Van Nostrand Reinhold Company, New York, 1979.

[Ben86] Jon L. Bentley. *Programming Pearls.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[BH85] A. Borodin and J. E. Hopcroft. Routing, Merging and Sorting on Parallel Models of Computation. *Journal of Computer and System Sciences*, 30:130–145, 1985.

[Bil89] Gianfranco Bilardi. Some Observations on Models of Parallel Computation. In *Proceedings of the NSF - ARC Workshop on Opportunities and Constraints of Parallel Computing ([San89a])*, pages 11–13, 1989.

[Bir79] Graham M. Birtwistle. *DEMOS — A System for Discrete Event Modelling on Simula.* The MacMillan Press Ltd., London, 1979.

[Boo87] Grady Booch. *Software Engineering With ADA.* The Benjamin/Cummings Publishing Company, Inc., 1987.

[CF90] David Cann and John Feo. SISAL versus FORTRAN: A Comparison Using the Livermore Loops. In *Proceedings of SUPERCOMPUTING'90, New York, November 12–16*, pages 626–636, 1990.

[CG88] Nicholas Carriero and David Gelernter. Applications experience with Linda. In *Proc. ACM/SIGPLAN Symp. on Parallel Programming*, 1988.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design,* A Foundation. Addison-Wesley Publishing Company, Reading, Massachusetts, 1988.

[Col86] Richard Cole. Parallel Merge Sort. In *Proceedings of 27th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 511–516, 1986.

[Col87] Richard Cole. Parallel Merge Sort. Technical Report 278, Computer Science Department, New York University, March 1987.

[Col88] Richard Cole. Parallel Merge Sort. *SIAM Journal on Computing,* 17(4):770–785, August 1988.

[Col89] Richard Cole. The APRAM: Incorporating Asynchrony into the PRAM Model. In *Proceedings of the NSF - ARC Workshop on Opportunities and Constraints of Parallel Computing ([San89a]),* pages 25–28, 1989.

[Col90] Richard Cole. New York University. Private communication, 16 March and 27 April 1990, 1990.

[Coo74] S. A. Cook. An Observation on Time-Storage Tradeoff. *Journal of Computer and System Sciences*, 9(3):308–316, 1974.

[Coo81] S. A. Cook. Towards a complexity theory of synchronous parallel computation. *L'Enseignement Mathematique*, XXVII:99–124, 1981.

[Coo83] Stephen A. Cook. An Overview of Computational Complexity. *Communications of the ACM*, 26(6):401–408, June 1983. Turing Award Lecture.

[Coo84] Stephen A. Cook. The Classification of Problems which have Fast Parallel Algorithms. In *Foundations of Computation Theory, Proceedings of the 1983 International FCT-Conference, Borgholm, Sweden, August 1983, Lecture Notes in Computer Science no. 158*, pages 78–93, Berlin, 1984. Springer Verlag.

[Coo85] Stephen A. Cook. A Taxonomy of Problems with Fast Parallel Algorithms. *Inform. and Control*, 64:2–22, 1985. This is a revised version of [Coo84].

[DeC89] Angel L. DeCegama. *The technology of parallel processing, Parallel processing architectures and VLSI hardware, Volume I.* Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1989.

[Dem85] Howard B. Demuth. Electronic Data Sorting. *IEEE Transactions on Computers*, C-34(4):296–310, April 1985.

[DH87] Hans Petter Dahle and Per Holm. SIMULA User's Guide for UNIX. Technical report, Lund Software House AB, Lund, Sweden, 1987.

[DKM84] C. Dwork, P. C. Kanellakis, and J. C. Mitchell. On the Sequential Nature of Unification. *Journal of Logic Programming*, 1(1):35–50, 1984.

[DLR79] D. Dobkin, R. J. Lipton, and S. Reiss. Linear programming is log–space hard for P. *Information Processing Letters*, 8:96–97, 1979.

[Dun90] Ralph Duncan. A Survey of Parallel Computer Architectures. *IEEE Computer*, 23(2):5–16, February 1990.

[Fei88] Dror G. Feitelson. *Optical Computing, A Survey for Computer Scientists.* The MIT Press, London, England, 1988.

[Fis88] David C. Fisher. Your Favorite Parallel Algorithms Might Not Be as Fast as You Think. *IEEE Transactions on Computers*, 37(2):211–213, 1988.

[Flo62] R. W. Floyd. Algorithm 97, shortest path. *Communications of the ACM*, 5:345, 1962. The algorithm is based on the theorem in [War62].

[Fly66] Michael J. Flynn. Very High-Speed Computing Systems. In *Proceedings of the IEEE*, volume 54, pages 1901–1909, December 1966.

[Fre86a] Karen A. Frenkel. Complexity and Parallel Processing: An Interview With Richard Karp. *Communications of the ACM*, 29(2):112–117, February 1986.

[Fre86b] Karen A. Frenkel. Special Issue on Parallelism. *Communications of the ACM*, 29(12):1168–1169, December 1986.

[FT90] Ian Foster and Stephen Taylor. *Strand: New Concepts in Parallel Programming.* Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1990.

[FW78] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *Proceedings of the 10'th ACM Symposium on Theory of Computing (STOC)*, pages 114–118. ACM, NewYork, May 1978.

[Gel86] David Gelernter. Domesticating Parallelism. *IEEE Computer*, pages 12–16, August 1986. Guest Editor's Introduction.

[Gib89] Phillip B. Gibbons. Towards Better Shared Memory Programming Models. In *Proceedings of the NSF - ARC Workshop on Opportunities and Constraints of Parallel Computing ([San89a])*, pages 55–58, 1989.

[GJ79] Michael R. Garey and David S. Johnson. *COMPUTERS AND INTRACTABILITY, A Guide to the Theory of NP–Completeness.* W. H. Freeman and Co., New York, 1979. If you find this book difficult, you might try [Wil86] or [RS86].

[GMB88] John L. Gustafson, Gary R. Montry, and Robert E. Benner. Development of Parallel Methods for a 1024-Processor Hypercube. *SIAM Journal on Scientific and Statistical Computing*, 9(4):609–638, July 1988.

[Gol77] L. M. Goldschlager. The monotone and planar circuit value problems are log space complete for P. *SIGACT News*, 9(2):25–29, 1977.

[GR88] Alan Gibbons and Wojciech Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, Cambridge, 1988.

[GS89] Mark Goldberg and Thomas Spencer. A new parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 18(2):419–427, April 1989.

[GSS82] Leslie M. Goldschlager, Ralph A. Shaw, and John Staples. The maximum flow problem is log space complete for P. *Theor. Comput. Sci*, 21(1):105–111, October 1982.

[Gup90] Rajiv Gupta. Loop Displacement: An Approach for Transforming and Scheduling Loops for Parallel Execution. In *Proceedings of SUPERCOMPUTING'90, New York, November 12–16*, pages 388–397, 1990.

[Gus88] John L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31(5):532–533, May 1988.

[Hag91] Marianne Hagaseth. Very fast pram sorting algorithms (preliminary title). Master's thesis, Division of Computer Systems and Telematics, The Norwegian Institute of Technology, February 1991. In preparation.

[Hal85] Robert Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages*, October 1985.

[Har87] David Harel. *ALGORITHMICS, The Spirit of Computing.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1987.

[HB84] Kai Hwang and Fayé A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill Book Company, New York, 1984.

[Hil87] W. Daniel Hillis. The Connection Machine. *Scientific American*, 256(6):86–93, June 1987.

[Hil90] W. Daniel Hillis. The Fastest Computers. Keynote Address at the SUPERCOMPUTING'90 conference, New York, 13 November, 1990., 1990. Notes from the presentation.

[HJ88] R. W. Hockney and C. R. Jesshope. *Parallel Computers 2 : architecture, programming and algorithms, 2nd edition.* Adam Hilger, Bristol, 1988.

[HM89] David P. Helmbold and Charles E. McDowell. Modeling Speedup(n) greater than n. In *Proceedings of the 1989 International Conference on Parallel Processing, vol. III*, pages 219–225, 1989.

[Hoa62] C. A. R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.

[Hop87] John E. Hopcroft. Computer Science: The Emergence of a Discipline. *Communications of the ACM*, 30(3):198–202, March 1987. Turing Award Lecture.

[HS65] J. Hartmanis and R. E. Stearns. On the computational complexity of algorithms. *Trans. Amer. Math. Soc.*, 117:285–306, May 1965.

[HS86] W. Daniel Hillis and Guy L. Jr. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[HT87] Per Holm and Magnus Taube. SIMDEB User's Guide for UNIX. Technical report, Lund Software House AB, Lund, Sweden, 1987.

[Hud86] Paul Hudak. Para-Functional Programming. *IEEE Computer*, pages 60–69, August 1986.

[HWG89] Michael Heath, Patrick Worley, and John Gustafson. Once Again, Amdahl's Law, *and* Author's Response. *Communications of the ACM*, 32(2):262–264, February 1989. Technical correspondance from Heath and Worley with the response from Gustafson.

[JL77] N. Jones and W. Laaser. Complete problems for deterministic polynomial time. *Theoretical Computer Science*, 3:105–117, 1977.

[Kan87] Paris C. Kanellakis. Logic Programming and Parallel Complexity. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 14, pages 547–585. Morgan Kaufmann Publishers, Inc., Los Altos, California, 1987.

[Kar86] Richard M. Karp. Combinatorics, Complexity and Randomness. *Communications of the ACM*, 29(2):98–109, February 1986. Turing Award Lecture.

[Kar89] Richard M. Karp. A Position Paper on Parallel Computation. In *Proceedings of the NSF - ARC Workshop on Opportunities and Constraints of Parallel Computing ([San89a])*, pages 73–75, 1989.

[Kha79] L. G. Khachian. A polynomial time algorithm for linear programming. *Dokl. Akad. Nauk SSSR*, 244(5):1093–1096, 1979. In Russian. Translated to English in *Soviet. Math. Dokl.* , vol. 20, 191–194.

[KM90] Ken Kennedy and Kathryn S. McKinley. Loop Distribution with Arbitrary Control Flow. In *Proceedings of SUPERCOMPUTING'90, New York, November 12–16*, pages 407–416, 1990.

[Knu73] D. E. Knuth. *The Art of Computer Programming: Vol. 3 Sorting and Searching.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1973.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall Inc, New Jersey, 1978.

[KR88] Richard M. Karp and Vijaya Ramachandran. A Survey of Parallel Algorithms for Shared-Memory Machines. Technical Report 88/408, Computer Science Division, University of California, Berkeley, March 1988.

[Kru56] J. B. Kruskal. On the shortest spanning subtree of a graph and the travelling salesman problem. *Proceedings of the American Mathematical Society*, 7:48–55, 1956.

[Kru89] Clyde P. Kruskal. Efficient Parallel Algorithms: Theory and Practice. In *Proceedings of the NSF - ARC Workshop on Opportunities and Constraints of Parallel Computing ([San89a])*, pages 77–79, 1989.

[Kuc77] David J. Kuck. A Survey of Parallel Machine Organization and Programming. *ACM Computing Surveys*, 9(1):29–59, March 1977.

[Kun82] H. T. Kung. Why Systolic Architectures? *IEEE Computer*, pages 37–46, January 1982.

[Kun89] H. T. Kung. Computational models for parallel computers. In *Scientific applications of multiprocessors*. Prentice Hall, Engelwood Cliffs, New Jersey, 1989.

[Lad75] R. E. Ladner. The circuit value problem is log space complete for P. *SIGACT News*, 7(1):18–20, 1975.

[Lan88] Asgeir Langen. Parallelle programmeringsspråk. Master's thesis, Division of Computer Systems and Telematics, The Norwegian Institute of Technology, The University of Trondheim, Norway, 1988. In Norwegian.

[Law76] Eugene L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, New York, 1976.

[Lei84] Tom Leighton. Tight Bounds on the Complexity of Parallel Sorting. In *Proceedings of the 16th Annual ACM Symposium on Theory Of Computing (May)*, pages 71–80. ACM, New York, 1984.

[Lei89] Tom Leighton. What is the Right Model for Designing Parallel Algorithms? In *Proceedings of the NSF - ARC Workshop on Opportunities and Constraints of Parallel Computing ([San89a])*, pages 81–82, 1989.

[Lub89] Boris D. Lubachevsky. Synchronization barrier and tools for shared memory parallel programming. In *Proceedings of the 1989 International Conference on Parallel Processing, vol. II*, pages 175–179, 1989.

[Lun90] Stephen F. Lundstrom. The Future of Supercomputing, The Next Decade and Beyond. Technical Report 12, November 12, 1990, PARSA, 1990. Documentation from tutorial on SUPERCOMPUTING'90.

[Mag89] Bruce Maggs. Beyond parallel random-access machines. In *Proceedings of the NSF - ARC Workshop on Opportunities and Constraints of Parallel Computing ([San89a])*, pages 83–84, 1989.

[Man85] Heikki Manilla. Measures of Presortedness and Optimal Sorting Algorithms. *IEEE Transactions on Computers*, C-34(4), April 1985.

[MB90] Peter Moller and Kallol Bagchi. A Design Scheme for Complexity Models of Parallel Algorithms. In *Proceedings of the 4'th IEEE Annual Parallel Processing Symposium, California*, 1990.

[McG89] Robert J. McGlinn. A Parallel Version of Cook and Kim's Algorithm for Presorted Lists. SOFTWARE—PRACTICE AND EXPERIENCE, 19(10):917–930, October 1989.

[MH89] Charles E. McDowell and David P. Helmbold. Debugging Concurrent Programs. *ACM Computing Surveys*, 21(4):593–622, 1989.

[MK84] J. Mikloško and V. E. Kotov, editors. *Algorithms, Software and Hardware of Parallel Computers.* Springer Verlag, Berlin, 1984.

[MLK83] G. Miranker, Tang L., and Wong C. K. A "Zero-Time" VLSI Sorter. *IBM J. Res. Develop.*, 27(2):140–148, March 1983.

[Moh83] Joseph Mohan. Experience with Two Parallel Programs Solving the Traveling Salesman Problem. In *Proceedings of the 1983 International Conference on Parallel Processing*, pages 191–193. IEEE, New York, 1983.

[MV84] Kurt Mehlhorn and Uzi Vishkin. Randomized and Deterministic Simulations of PRAMs by Parallel Machines with Restricted Granularity of Parallel Memories. *Acta Informatica*, 21:339–374, 1984.

[Nat89] Lasse Natvig. Comparison of Some Highly Parallel Sorting Algorithms With a Simple Sequential Algorithm. In *Proceedings of NIK 89, Norsk Informatikk Konferanse (Norwegian Informatics Conference), Stavanger, November*, pages 35–49, 1989.

[Nat90a] Lasse Natvig. Investigating the Practical Value of Cole's $O(\log n)$ Time CREW PRAM Merge Sort Algorithm. In *Proceedings of The Fifth International Symposium on Computer and Information Sciences (ISCIS V), Cappadocia, Nevsehir, Turkey, October 30 – November 2*, pages 627–636, 1990.

[Nat90b] Lasse Natvig. Logarithmic Time Cost Optimal Parallel Sorting is *Not Yet* Fast in Practice! In *Proceedings of SUPERCOMPUTING'90, New York, November 12–16*, pages 486–494, 1990.

[Nat90c] Lasse Natvig. Parallelism Should Make Programming Easier! Presentation at the Workshop On The Understanding of Parallel Computation, Edinburgh, July 11–12, 1990.

[OC88] Rodney R. Oldehoeft and David C. Cann. Applicative Parallelism on a Shared-Memory Multiprocessor. *IEEE Software*, pages 62–70, 1988.

[Par86] D. Parkinson. Parallel efficiency can be greater than unity. *Parallel Computing*, pages 261–262, 1986.

[Par87] Ian Parberry. *Parallel Complexity Theory.* John Wiley and Sons, Inc., New York, 1987.

[Pat87] M. S. Paterson. Improved sorting networks with $O(\log n)$ depth. Technical report, Dept. of Computer Science, University of Warwick, England, 1987. Res. Report RR89.

[Pip79] N. Pippenger. On simultaneous resource bounds (preliminary version). In *Proc. 20th IEEE Foundations of Computer Science*, pages 307–311, 1979.

[Pol88] Constantine D. Polychronopoulos. *PARALLEL PROGRAMMING AND COMPILERS.* Kluwer Academic Publishers, London, 1988.

[Poo87] R. J. Pooley. *An Introduction to Programming in SIMULA.* Blackwell Scientific Publications, Oxford, England, 1987.

[QD84] Michael J. Quinn and Narsingh Deo. Parallel Graph Algorithms. *ACM Computing Surveys*, 16(3):319–348, September 1984.

[Qui87] Michael J. Quinn. *Designing Efficient Algorithms for Parallel Computers.* McGraw-Hill Book Company, New York, 1987.

[RBJ88] Abhiram G. Ranade, Sandeep N. Bhatt, and S. Lennart Johnsson. The Fluent Abstract Machine. In *Proceedings of the Fifth MIT Conference on Advanced Research in VLSI*, pages 71–94, 1988.

[Rei85] J. Reif. Depth first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, 1985.

[Rei86] Rüdiger K. Reischuk. Parallel Machines and their Communication Theoretical Limits. In *STACS 86*, pages 359–368, 1986.

[Ric86] D. Richards. Parallel sorting—a bibliography. *ACM SIGACT News*, pages 28–48, 1986.

[RS86] V. J. Rayward-Smith. *A First Course in Computability.* Blackwell Scientific Publications, Oxford, 1986.

[RS89] John H. Reif and Sandeep Sen. Randomization in parallel algorithms and its impact on computational geometry. In *LNCS 401*, pages 1–8, 1989.

[Rud90] Larry Rudolph. Hebrew University, Israel. Private communication, 15 May and 17 June 1990, 1990.

[RV87] John H. Reif and Leslie G. Valiant. A Logarithmic Time Sort for Linear Size Networks. *Journal of the ACM*, 34(1):60–76, January 1987.

[Sab88] Gary W. Sabot. *The Paralation Model* Architecture-Independent Parallel Programming. The MIT Press, Cambridge, Massachusetts, 1988.

[Sag81] Carl Sagan. *Kosmos.* Universitetsforlaget, 1981. Norwegian edition.

[San88] Sandia National Laboratories. Scientists set speedup record, overcome barrier in parallel computing. News Release from Sandia National Laboratories, Albuquerque, New Mexico, March 1988.

[San89a] Jorge L. C. Sanz, editor. *Opportunities and Constraints of Parallel Computing.* Springer-Verlag, London, 1989. Papers presented at the NSF - ARC Workshop on Opportunities and Constraints of Parallel Computing, San Jose, California, December 1988. (ARC = IBM Almaden Research Center, NSF = National Science Foundation).

[San89b] Jorge L. C. Sanz. The Tower of Babel in Parallel Computing. In *Proceedings of the NSF - ARC Workshop on Opportunities and Constraints of Parallel Computing ([San89a])*, pages 111–115, 1989.

[SG76] Sartaj Sahni and Teofilo Gonzales. P-Complete Approximation Problems. *Journal of the ACM*, 23(3):555–565, July 1976.

[Sha86] Ehud Shapiro. Concurrent Prolog: A Progress Report. *IEEE Computer*, pages 44–58, August 1986.

[SIA90] SIAM. Both Gordon Bell Prize Winners Tackle Oil Industry Problems, May 1990. SIAM = The Society for Industrial and Applied Mathematics.

[SN90] Xian-He Sun and Lionel M. Ni. Another View on Parallel Speedup. In *Proceedings of SUPERCOMPUTING'90, New York, November 12–16*, pages 324–333, 1990.

[Sni89] Marc Snir. Parallel Computation Models—Some Useful Questions. In *Proceedings of the NSF - ARC Workshop on Opportunities and Constraints of Parallel Computing ([San89a])*, pages 139–145, 1989.

[Sny88] Lawrence Snyder. A Taxonomy of Synchronous Parallel Machines. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 281–285, 1988.

[Spi86] Paul Spirakis. The Parallel Complexity of Deadlock Detection. In *Mathematical Foundations of Computer Science 1986, Bratislava, Czechoslovakia, Lecture Notes in Computer Science no. 233*, pages 582–593, 1986.

[Ste90] Guy L. Steele. Making Asynchronous Parallelism Safe for the World. In *Proceedings of POPL-90*, pages 218–227, 1990.

[SV81] Yossi Shiloach and Uzi Vishkin. Finding the Maximum, Merging and Sorting in a Parallel Computation Model. *Journal of Algorithms*, 2:88–102, 1981.

[SV84] Larry Stockmeyer and Uzi Vishkin. Simulation of Parallel Random Access Machines by Circuits. *SIAM Journal on Computing*, 13(2):409–422, May 1984.

[Tar72] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[Tha90] Peter Thanisch. Department of Computer Science, University of Edinburgh. Private communication, 12 July 1990, 1990.

[Tiw87] Prasoon Tiwari. Lower Bounds on Communication Complexity in Distributed Computer Networks. *Journal of the ACM*, 34(4):921–938, October 1987.

[Upf84] Eli Upfal. A Probabilistic Relation Between Desirable and Feasible Models of Parallel Computation. In *Proceedings of the 16'th ACM Symposium on Theory of Computation. (STOC)*, pages 258–265, 1984.

[Val75] L. Valiant. Parallelism in comparison problems. *SIAM Journal on Computing*, 4:348–355, 1975.

[Val90] Leslie G. Valiant. A Bridging Model for Parallel Computation. In *Proceedings of the 5th Distributed Memory Computer Conference, Charleston, California, April*, 1990. 20 pages.

[Vis89] Uzi Vishkin. PRAM Algorithms: Teach and Preach. In *Proceedings of the NSF - ARC Workshop on Opportunities and Constraints of Parallel Computing ([San89a])*, pages 161–163, 1989.

[War62] Stephen Warshall. A Theorem on Boolean Matrices. *Journal of the ACM*, 9:11–12, 1962. This and [Flo62] are commonly considered as the source for the famous Floyd-Warshall algorithm.

[Wil64] J. W. J. Williams. Heapsort (algorithm 232). *Communications of the ACM*, 7(6):347–348, 1964.

[Wil86] Herbert S. Wilf. *Algorithms and Complexity.* Prentice Hall International, Inc., Englewood Cliffs, New Jersey, 1986.

[Wir76] Niklaus Wirth. *Algorithms + Data Structures = Programs.* Prentice Hall, Englewood Cliffs, New Jersey, 1976.

[WW86] K. Wagner and Wechsung. *Computational Complexity.* D. Reidel Publishing Company, Dordrecht, Holland, 1986.

[Wyl79] J. C. Wyllie. *The Complexity of Parallel Computations.* PhD thesis, Dept. of Computer Science, Cornell University, 1979.

[Zag90] Marcoz Zagha. Carnegie Mellon University. Private communication, 16 Nov 1990, 1990.

[ZG89] Xiaofeng Zhou and John Gustafson. Bridging the Gap Between Amdahl's Law and Sandia Laboratory's Result, *and* Author's Response. *Communications of the ACM*, 32(8):1014–1016, August 1989. Technical correspondance from Zhou with the response from Gustafson.

Index