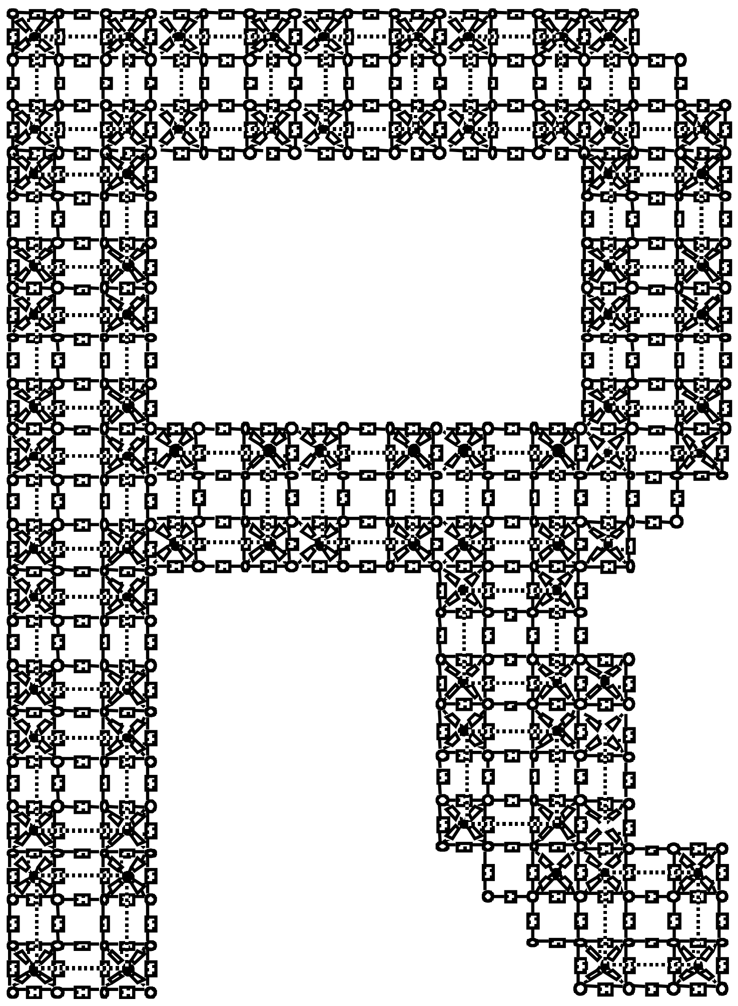
DESIGN CONSIDERATIONS FOR A PARALLEL REDUCTION MACHINE

Willem Vree



Design Considerations For A Parallel Reduction Machine

Design Considerations For A Parallel Reduction Machine

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de

Universiteit van Amsterdam

op gezag van de Rector Magnificus

prof. dr. S. K. Thoden van Velzen,

in het openbaar te verdedigen in de Aula der Universiteit

(Oude Lutherse Kerk, ingang Singel 411, hoek Spui),

op woensdag 5 december 1989 te 12.00 uur

door

Willem Gerard Vree

geboren te Amsterdam

Amsterdam 1989

Promotor: Prof. Dr. L.O. Hertzberger

Faculteit: Wiskunde en Informatica

Cover: Ellen Vree

Printed by Sneldruk Enschede, October 1989

ter nagedachtenis aan mijn vader

Table of Contents

Preface....xi

Acknowledgement....xii

Chapter I 13

1 Parallelism in computer architectures....15

Chapter II 25

2. Implementation of reduction....27

Chapter III 41

3 Architectures for parallel reduction....43

References....67

Chapter IV 71

A Coarse Grain Parallel Architecture for Functional Languages....73

abstract....73

1. Introduction....73

2. Separation of reduction and parallelism....74

3. The architecture of APERM....78

4. Performance Measurements....81

5. Comparison and discussion of results....86

6. Acknowledgement....87

7. References....87

Chapter V 89

Parallel graph reduction for divide-and-conquer applications.
Part I - program transformations....91

Chapter VI 119

The Grain Size of Parallel Computations in a Functional Program....121

1. Introduction 121

2. Grain size enlargement....122

3. The physical problem....123

4. The first program....124

5. Transformation to the second version....126

6. Transformation to the third version....129

7. Data grouping....133

8. Conclusion....133

9. Acknowledgements....134

10. References....134

Chapter VII 135

Parallel Graph Reduction for Synchronous Process Networks....137

abstract....137

1 Introduction....137

2 Job model....139

3 Graph rewriting....140

4 Sequential graph reduction of process networks....142

5 Communication lifting....149

6 Using communication lifting....160

7 Conclusion....168

8 Acknowledgements....168

9 References....168

Appendix: Summary of communication lifting....171

Chapter VIII 175

Parallel graph reduction for divide-and-conquer applications.
Part II - program performance....177

Preface

It was around the beginning of 1984 when Henk Barendregt took the initiative to raise funds for a project exploring the feasibility of a parallel reduction machine. This initiative proved to be a very fruitful one with many consequences, amongst which are my decision to join the University of Amsterdam eventually in 1987 and the completion of this thesis, being the third in a row of theses born form the Dutch Parallel Reduction Machine Project.

When the Dutch Parallel Reduction Machine Project started in the late summer of 1984, I was employed as a scientific consultant at the computing division of the Dutch Water-Board. During one of the stimulating discussions I had with Bob Hertzberger in that time, he drew my attention to parallel reduction machines in general and the reduction machine project in particular. Several months later I was allowed to take part in the project on behalf of both the Water-Board and the University of Amsterdam.

The interest of the Dutch Water-Board was mainly in the development of large parallel simulation programs, whereas the University of Amsterdam contributed to the project in the area of parallel computer architecture. The present thesis that arose form this mixture of interests covers the design of a specialised parallel reduction machine where programming considerations play an important role.

Viewed in retrospect, the development of a parallel reduction machine has been very successful, considering the amount of criticism from both scientific and industrial side that had to be faced during the project. In the past four years much experience has been gained and a vast amount of knowledge in the area of reduction and its implementation has been acquired in the Computer Systems Department of the University of Amsterdam. Based on the results as reported in this thesis we expect in the next three to four years to design an improved version of the present experimental machine in which new problem areas concerning the exploitation of parallelism will be explored.

Hilversum, October 1989.

Acknowledgement

First of all I wish to thank Bob Hertzberger for offering me the possibility to complete a thesis during the exigent duties of my current position and for providing sufficient resources to set up a group of specialists in the area of reduction machines.

I am very grateful to Henk Barendregt for having started and sustained the project and for the many insights in fundamental matters (not only reduction or lambda calculus) he revealed.

In particular for the first years of the project I owe much to Theun Bruins. He continuously showed an inspiring interest in reduction machines and their applications, also on behalf of the Dutch Water-Board and he created the possibility for me to participate in the project.

Many thanks I am indepted to Pieter Hartel for spending so much time and effort in co-authoring two of the papers presented in this thesis. I also appreciate the fruitful discussions with Rutger Hofman and Koen Langendoen, which gave rise to the architectural considerations as exposed in chapter 3.4

I would like to express my gratitude to all members of the Dutch Parallel Reduction Machine Project for the many years of close collaboration.

Finally, I have to thank Ellen, Daniël and Susannah for enduring the many hours of my (mental or physical) absence while we ought to be together and for believing that the thesis would once be completed.

Chapter I

INTRODUCTION - Parallelism in computer architectures

1 Parallelism in computer architectures

Speed of computation is an essential property of computers. During the history of their existence the performance of computers has increased by many orders of magnitude, mainly due to constant improvements in electronic technology. Apart from employing faster technology it is also possible to raise the speed of computation by the exploitation of *parallelism*. Instead of speeding up a computation by increasing the speed of its basic sequential operations, it is sometimes possible to divide a computation into several independent parts that may be simultaneously computed on different machines. These simultaneous calculations can result in a considerable reduction of the execution time that was needed for the original computation, where the independent parts were evaluated one after the other. An interesting difference between the two approaches to increase computer performance is that theoretically there seems to be no restriction on the amount of parallelism that can be exploited, whereas sequential computation will eventually encounter fundamental physical limitations.

In the recent past the possibility to build parallel computers has become more and more appealing. On the one hand, switching speed of electronic components has increased to such a height that the speed of light poses severe limitations on the physical length of interconnections in a computer. On the other hand, integration technology of semiconductors will soon reach a point where a complex conventional computer architecture only occupies a fraction of the effective space on a silicon chip. One of the possibilities to fill up the available space on a silicon chip is to design regular parallel computers, where a basic processor and communication design can be copied as often as required. These architectures are often called transparent-, scalable- or *extensible architectures*.

Although there is no theoretical upper bound on the amount of parallelism that may be exploited in an extensible architecture, in practice the cost of communication may pose restrictions. In extensible architectures the time involved in communication between two arbitrary processors is not independent of the total number of processors. For a scalable communication design the cost of message transmission is proportional to the number of processors in the system. A useful measure with respect to the exploitability of parallelism is the grain-size of computations. The *grain-size* of a computation may be defined as the ratio between the computation cost and communication cost to perform the calculation. Traditionally, *fine-grain* and *coarse-grain* parallelism are distinguished. In a coarse-grain computation the computation cost is much higher than the communication cost while the inverse is true for fine-grain computations.

In the past, both fine-grain and coarse-grain parallelism have been frequently exploited in computer architectures. The use of pipe-lining in CPU's is an example of fine-grain parallelism. Small operations like instruction-fetch, decode, operand-fetch and instruction-execution are performed in parallel as often as possible.

Also coarse-grain parallelism is used in conventional architectures. For instance disk operations may be executed by specialized processors, while the CPU continues with other tasks. Multi-processor architectures have been used for twenty years now (e.g. Univac 1100 series), where the multiple CPU's execute independent sequential user-programs.

What is really new in todays research into parallel architectures, is the exploitation of massive parallelism. The increase in processing power envisaged by massively parallel architectures is at least two orders of magnitude compared to sequential execution on such an architecture.

The size and power consumption of computers impose a physical limit on the maximum achievable computational capacity. In massively parallel architectures processors have to consume less power and have to be smaller then in single processor designs. The restriction on size and power consumption dictates the use of relatively slow semiconductor technology in a massively parallel multi-processor architecture. Therefore the computational power of one processor in such an architecture is about one tenth or less of the power of a conventional single processor architecture of the same size as the multi-processor. The exploitation of parallelism should result in an ample compensation of this loss.

One of the major problems to effectively exploit the capacity of a massively parallel computer is how parallelism should be specified, or to put it differently: where does the parallelism come from? One of the possibilities is to choose a suitable programming language and to make use of all potential parallelism that is present in programs written in that language. Implementations based on this approach make use of the *implicit parallelism* present in the language.

For instance, *object oriented* programming languages [XER81] are based on a computational model that seems to offer good opportunities to exploit implicit parallelism. The objects in this model can be considered as the implicit units of parallel computation. Two large projects in the Netherlands supervised by Philips (DOOM [ODIJ85, ODIJ87] and PRISMA [BEE89]) are based on the use of an object oriented language to program a parallel machine. Both projects provide evidence that the choice of a suitable computational model in the form of a computer language by no means guarantees that massive parallelism is easily implemented. The main problem is that the specification of parallel computations in the object oriented model does not state anything about the grain-size of these computations. In both projects the application program has to provide information on the grain-size of parallel computations. For instance, the parallel relational data-base application on the PRISMA machine uses one-fragment managers (each of which controls part of a relation) as the grains of parallel computation.

It turns out that in the implementation of a language on a parallel architecture, there must be a precise match between the grain-size of parallel computations and the computation-communication performance ratio of the physical machine. If such a match can not be enforced, either a large part of the machine stays idle or, on the contrary, it is flooded with fine-grain computations. Especially the last possibility occurs in the object oriented model.

A similar problem has been reported by the Manchester dataflow group [GUR87]. A special throttle mechanism [RUG87] had to be incorporated in the data-flow machine to prevent overflow of the token store.

The grain-size of computations cannot be determined by a compiler, because grain-size is an undecidable property of a computation. Though it might be possible to devise certain heuristics to approximate the grain-size at compile time, no successful attempts in this direction have yet been reported up to the author's knowledge. Many research projects have chosen to base the implementation of parallelism on explicit grain-size information in the application programs. The programmer has to provide this information by, for instance, annotating coarse grain computations in the source text of the program. Implementations based on this approach are said to make use of *explicit coarse grain parallelism*. Whereas most former research projects were based on the use of implicit parallelism [MAG79, GUR87, DAR81, HUD85], several recent research projects included our own, have chosen only to exploit explicit coarse grain parallelism [EEK88, MCB87, KEL79, VRE88].

A similar trend can be observed in the development of the fifth generation computer project in Japan. In the beginning of this project it was believed that merely expressing a program in a logical language would provide a sufficient amount of potential parallelism for a fifth generation computer architecture. However, in recent practical parallel implementations parallel computations are restricted by programmer annotations.

When only explicit coarse grain parallelism is exploited the impact of a particular programming language on the implementation of a parallel architecture becomes less important. The grain-size of parallel computations is much more a property of the application program than of the programming language. When a certain algorithm is well suited for the annotation of coarse-grain parallel computations in one language, this situation will not change if the algorithm is recoded in another language. In most implementations grain-size information is not used during compilation, but is merely passed to the runtime system where it is used for triggering parallel computations. The programming language is transparent to the annotations of coarse-grain parallelism. So what is the advantage of the use of a particular language in this situation?

We use functional languages for our parallel architecture because of the excellent properties of these languages with respect to program transformations. We will show in this thesis that even parallel programs have to be modified considerably before annotation of explicit parallel calculations becomes possible. Additional modifications may be necessary to obtain the right balance between grain-size and architecture. These modifications can be elegantly performed by (formal) program transformations when the application program is expressed in a functional language. We have developed a number of program transformations that allow parallel application programs to be developed in a systematical way from (mathematical) specification to a well balanced version for a parallel architecture.

Using functional languages, program transformations can be more easily generalized, formalized and automated than using other languages. The main reason for this is that functional languages are referentially transparent. A language is called *referentially transparent* when multiple occurrences of a reference (variable) all denote the same value, independent of the place of the references in the program text (neglecting scope rules). Program transformations modify the syntactical appearance of the program. Therefore references may obtain a different position in the source text of the program. Referential transparency guarantees that the value of these references will not change.

Less formally expressed referential transparency means that once a variable obtains a certain value, references to that variable will always yield the same value. In particular this property forbids the use of an assignment operator, because it would then be possible to change the value of a variable by assignment, possibly resulting in different values for different occurrences of the same reference.

In principle, logical languages are also referential transparent. However, in contrast to functional languages, all implementations of logical languages contain non-referential transparent features (referential opaque features). The presence of these features (like the "cut-mark" in Prolog) make program transformations a lot more difficult.

Computer architectures that exploit all possibilities for parallel evaluation are called *fine-grain architectures*. In particular such architectures do not impose a lower bound on the grain size of parallel computations. In contrast, computer architectures that only exploit parallel computations with a grain-size above a certain threshold are called *coarse-grain architectures*. The trend towards the exploitation of coarse grain parallelism that is observed in the Japanese fifth generation project, the DOOM project, the PRISMA project and several reduction machine projects increases the relevance of our own research into coarse grain extensible architectures, in which from the beginning exploitation of parallelism was based on explicit annotation of coarse grain computations in the application program.

To show the viability of basic concepts underlying a parallel architecture, care must be taken with the measurements that are supposed to provide the evidence of successful operation of the architecture in question. For instance, in the area of compilation techniques for functional languages the efficiency of the proposed method is often demonstrated with performance results based on the "nfib" program [BRU87, JOH84, FAI87, MEY88]. These results cannot be considered of any real value, because the proposed compiler optimizations, which seem to work for such toy programs, may fail to produce the same results in case of larger application programs. The reason for a different behaviour on larger programs is that the optimization algorithms are based on undecidable properties of the source program, like the amount of sharing of expressions and the strictness of user-defined functions. The algorithms trying to derive information about these properties perform much better on simple programs than on complex ones.

For the same reason the use of toy programs to demonstrate the efficient exploitation of parallelism (either implicit or explicit), may produce unrealistic results. It is relatively easy to implement optimizations that produce good results for small programs.

Therefore we have based the performance analysis of our architecture on parallel programs of at least medium size. To support the construction of larger parallel applications we have developed two program transformation methods, corresponding to two basically different ways in which coarse grain parallelism may be obtained.

The first method, which we call data-partitioning, is applicable when a program specifies a coarse grain computation that may be split into several finer grains. The second method, which we call data-grouping, can be applied when an application consists of many fine grain computations that may be grouped into fewer but coarser grains. The data-partitioning method can handle divide-and-conquer algorithms, whereas the data-grouping method is used for programs written as networks of communicating processes.

How a given application program is transformed into a parallel version is illustrated in table 1. First, one has to determine if the application belongs to one of the two classes that we can deal with. If so, table 1 shows in which order transformations have to be applied. The table also indicates to what extent the transformations are formalized, how they are called and in which chapter they are described.

	data-grouping transformations (chapter 7)	formalized	data-partitioning transformations (chapter 5)	formalized
1)	communication lifting	yes	–	--
2)	job lifting	specialized	job lifting	yes
3)	grain size transformation	specialized	grain size transformation	yes
4)	own transformation	no	--	--

Table 1: transformations for data-grouping and data-partitioning

Although it appears from table 1 that step 2 and 3 of the data-grouping transformations are identical to the corresponding steps in the data-partitioning transformations, they are in fact specialized versions of the latter. A program resulting from a communication lifting transformation has such a specific form that specialized versions of the job-lifting and grain-size transformations are justified (These specialised versions are called the sandwich transformations in chapter 7).

We have applied the transformations of table 1 to a small set of realistic applications. The resulting parallel programs have been used to evaluate the performance of our architecture (see chapter 8). The data-partitioning transformations produced parallel versions of e.g. the fast Fourier transform, Wang's algorithm and a scheduling program. With data-grouping transformations we constructed a parallel version of a tidal model of the North Sea.

None of the transformations has been automated so far, although the formal descriptions suggest that a substantial part of the transformations may be implemented as tools assisting the programmer to create parallel programs. The construction of correctness proofs for the proposed transformations is a possible subject for future research.

To analyse the performance of large application runs we have developed the method of *hybrid simulation*. This method allows to obtain realistic performance results for large application programs without having to go through a complete implementation of a parallel architecture. Lower layers of the architecture (i.e. layers close to the hardware) are actually implemented on the target machine. A realistic simulation of these layers would take an unproportionally large amount of computing time, effectively shutting off the possibility to measure large applications.

Higher layers are not implemented on the target architecture. Instead they are simulated on a general purpose sequential host computer. Data obtained by the simulation is used as input for the implementation of the lower layers on the target architecture. Measurements obtained from the target machine are fed back into a performance model of the overall architecture. Figure 1 shows the flow of information for hybrid simulation. The simulation executes the application program and computes an execution profile that contains all data relevant to the lower level layers. The performance model uses both the execution profile and the data obtained from measurements on the target architecture.

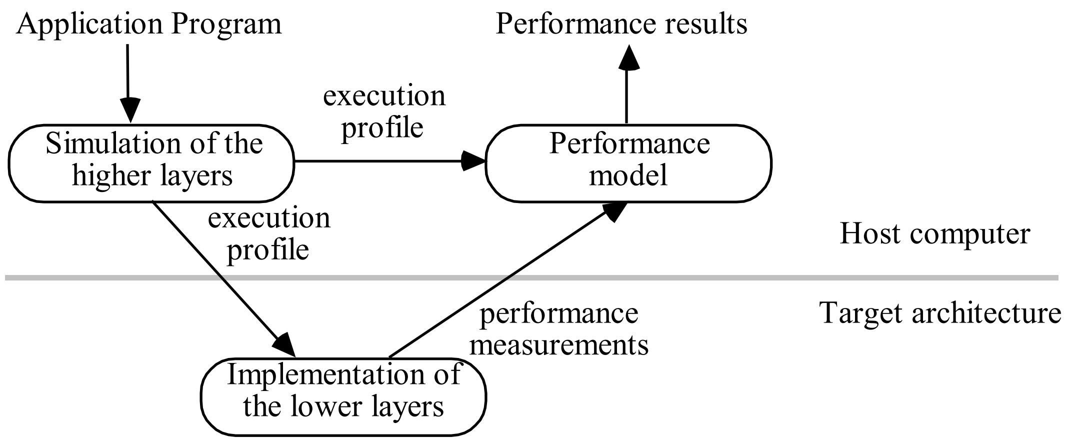


Figure 1: Hybrid Simulation

Hybrid simulation is applied in chapter 8 to evaluate the performance of our parallel architecture described in chapter 3 and 4.

In this thesis the exploitation of parallelism in computer architectures is studied in relation to one specific computational model: the reduction model. This model has been chosen because of the outstanding properties with respect to program transformation and parallel evaluation strategies. Although logical languages are claimed to have the same properties, parallel evaluation strategies turn out to be much more difficult to implement than in case of reduction. We have already mentioned the practical drawbacks of logical languages concerning program transformations.

The work reported in this thesis provides evidence that coarse-grain parallel reduction is applicable to a wide range of application programs. We present an efficient mapping of several parallel functional programs onto a coarse-grain architecture. In addition we show that these programs can be developed in a systematical way using several transformations. By means of hybrid simulation, measurements are obtained for the developed programs on our experimental parallel reduction machine. Based on these results we present an analysis of the performance of this architecture covering the whole implementation trajectory.

To reduce the amount of work involved in studying the implementation of parallel reduction, two a-priory restrictions have been made:

At first, it was decided only to consider coarse grain parallelism. The reasons for this choice are of a practical nature. At the University of Amsterdam already much experience had been gained with coarse grain parallel architectures in the area of data-acquisition and filtering for high energy physics experiments. Because of the close collaboration with the physics department, it was relatively easy to construct a coarse grain parallel architecture based on the use of dual-ported memories (see chapter 3 and 4).

The second decision concerns the separation of sequential reduction and the control of parallelism. The use of coarse-grain "strict argument" parallelism allows the design of a parallel reduction model that is valid for all possible implementations of sequential reduction (see chapter 5). The issues covered by this thesis are centred around this coarse-grain parallel reduction model. The model requires annotation of parallel coarse-grain expressions in application programs, but it does not specify how these expressions have to be (sequentially) reduced. Any sequential implementation of reduction can be plugged into the parallel model. The decision appeared to be fruitful, because much progress has been made in the implementation of sequential reduction. These results can be directly used in the implementation of our reduction model.

1.1 Synopsis

Chapter 2 provides a short introduction into the reduction model of computation. Next, an overview is presented of the various ways in which reduction can be implemented. The overview provides the reader with some basic information about the implementation of reduction that will be used in the subsequent chapter on parallel reduction architectures. The subject of implementing sequential reduction is not further pursued in this thesis.

Chapter 3 presents two models that will be used to describe respectively abstract- and concrete architectures. The concepts and terminology of these two models together with the concepts related to the implementation of reduction (chapter 2) constitute a framework for the comparison of parallel reduction architectures. Chapter 3 contains a comparative description of some parallel reduction machines, within the given framework. Our own architecture, the Amsterdam Parallel Experimental Reduction Machine (APERM) is contrasted with the described architectures by highlighting the major differences and similarities.

Chapter 4 presents a more detailed description of APERM. Special attention is paid to the data-communication support based on dual ported memories. The advantage of the architecture with respect to the minimization of communication cost is discussed.

A coarse grain parallel reduction model is proposed in chapter 5. The grains of parallel evaluation are called jobs. The model is referred to as the job-based reduction model. The remainder of chapter 5 is devoted to the mapping of divide-and-conquer algorithms onto the job-based reduction model. Two program transformations are proposed to achieve this goal.

In chapter 6 a moderately sized application program is developed in a functional language. It is shown that a mathematical model of the tides in the North Sea can be systematically transformed into a coarse grain parallel functional program. The resulting program looks like a network of communicating processes. The transformations developed in chapter 5 to map divide-and-conquer algorithms onto the job-based reduction model appear to be insufficient to deal with process networks.

In chapter 7 additional transformations are proposed to widen the class of application programs that can profit from parallel evaluation in our reduction model. Also applications written as (functional) process networks are now included. As an example of these transformations the tidal simulation of chapter 6 is mapped onto the job-based reduction model. An additional application of a simulation of digital hardware is included in chapter 7, to show that also a collection of fine-grain communicating processes can be mapped onto the job-model.

Chapter 8 shows that the job-based reduction model can be efficiently mapped onto the APERM architecture. Several aspects of the abstract- and concrete architecture of APERM are discussed. Amongst these are the use of execution profiles of application programs in loadbalancing decisions and a presentation of the abstract instruction set concerning process-management and communication. Next, a simple performance model for APERM is developed and is applied to the results of hybrid simulation. In this way the performance of the application programs developed in the chapters 5, 6 and 7 is evaluated.

Chapter II

INTRODUCTION - Implementation of reduction

2. Implementation of reduction

Most computer architectures are based on a computational model proposed by Alan Turing in the 1930's. The essential property of the Turing model is that computation is performed by a sequence of commands, that manipulate the state of the computation. Programming languages based on the Turing model are often called *imperative languages*. The word "imperative" is inspired by the property that the abstract machine provided by the language is "instructed" by a sequence of commands. Computer architectures based on the Turing model are often called *Von Neumann* machines, named after Johannes von Neumann, who replaced the infinite sequential-access memory of the Turing model by a random-access memory, containing both data and program.

In 1936 Alan Turing proved that his model is equivalent to the lambda calculus of Alonzo Church, in the sense that both models describe the same class of functions. Turing and Church tried to mould the intuitive notion of a "computable" function into concrete. The conjecture is that every computable function can be expressed in the lambda calculus. However, the intuitive notion of "computability" leaves the possibility open that somebody might come along and compute a function that cannot be expressed in the lambda calculus. The inverse of the conjecture, namely that every function expressed in the lambda calculus can be computed, is the basis of computers.

Programming languages can be viewed as "syntactically sugared" versions of the underlying computational model. No extra power is added by a programming language, only the ease of expressing frequently occurring problems is increased. Languages based on the lambda calculus are called *functional languages* and a program written in a functional language is called a *functional program*. From the equivalence of the models of Turing and Church it follows that it is always possible to translate a functional program into an equivalent imperative program and vice versa. Recently it has been shown that the compilation of functional programs into imperative code can produce very efficient results [JOH84, FAI87].

Although the models of Turing and Church are equivalent, they are very different in nature. For instance, the lambda calculus is not "imperative". There is no sequence of commands to be executed, but merely an expression that has to be reduced (the computational mechanism of the lambda calculus is referred to as *reduction*). Computational models like the lambda calculus, without the notion of a state that changes during time, are called *declarative*. In contrast to the imperative model of Turing, a computation in such models is "declared" in a more or less static way. Programming languages based on a declarative model are referential transparent (see chapter 1). Once a reference denotes a certain value, there is no (assignment) mechanism that can change this value any more. All copies of the reference will denote the same value during evaluation of the program.

Another example of a declarative computational model is the interpretation of *Horn-clauses* by a computational mechanism called *resolution* [ROB65]. Languages based on Horn-clauses are called logical languages and the resolution mechanism is often referred to as *inference.*

In this thesis we confine our attention to the reduction model of computation in relation to hardware- and software implementation aspects of parallel architectures.

2.1 Parallelism in functional programs

The development of parallel programs in the imperative model is relatively difficult. Global state information that is present in imperative programs has to be distributed in some way amongst the parallel computations. After distribution, all computations still have the possibility to change any part of the global state. The programmer has to include explicit communication and synchronization instructions in the program to accomplish global state changes. In general, both the division of the global state and the insertion of communication instructions are no easy tasks.

For functional programs the transition from sequential to parallel programs is less complicated. No special language elements, like communication primitives in imperative languages, are needed. This is because parallelism is *implicitly* present in a functional program. An example of a functional expression may clarify this point:

```
        (3 + 4) * (5 - 2)
```

In the example the sub-expressions *(3 + 4)* and *(5 - 2)* may be evaluated in parallel. Although the example only shows a simple expression, the observed independence of subexpressions holds in general: Due to referential transparency, sub-expressions in functional programs are independent and may be evaluated in parallel. This parallelism is called *implicit*, because there is no language-construct that indicates which expressions are valid candidates for parallel evaluation.

Implicit parallelism can be exploited, either by making it explicit in the source text of the program or by trying to detect it automaticly. In the first case the programmer is required to provide annotations, marking sub-expressions that can be safely computed in parallel. In the second case a compiler or a run-time system takes care of the parallelism. In both cases the correctness of the program is not affected by the transition from sequential to parallel evaluation.

With respect to massively parallel computer architectures, most functional programs do not exhibit a sufficient amount of implicit parallelism, or the parallelism is too fine grained to be exploited. It is necessary to transform most application programs to make them suited for efficient evaluation on a parallel architecture. The additional complexity of these transformations should be compared with the effort that is necessary to design parallel programs in the imperative style. Because of the referential transparency, declarative languages and especially functional languages are well suited for the development of program transformations. Chapters 5, 6 and 7 discuss transformation techniques that can be used to obtain efficient parallel functional programs for a broad class of applications.

2.2 Reduction

The reduction principle can be viewed as replacing expressions by other expressions, based on certain rules. In this view a functional program consists of a set of reduction rules and a main expression. During the evaluation of the program, the main expression is repeatedly rewritten, replacing sub-expressions as prescribed by the rules, until no more replacements are possible. The expression is then said to be in *normal* form and represents the final answer of the program. The action of performing one replacement is called a *reduction step*. An expression that can be replaced is called a *redex* (reducible expression).

The order in which the reduction rules are applied is known as the reduction *strategy*. For instance one may have sequential or parallel strategies. An important property of practical reduction systems is the fact that all reduction strategies yield the same answer, provided the reduction process terminates. Reduction systems having this property are called *Church-Rosser* or *confluent*. In principle the Church-Rosser property allows any degree of parallelism during the evaluation of a program. The possibility of non-termination spoils the perspective of having the complete freedom to rewrite all reducible expressions in parallel. Some of these expressions might fail to terminate, whereas they will later appear not to be needed in the construction of the final answer. If a parallel architecture happens to assign non-terminating expressions to all of the available processing power, the program will never terminate, while there still exists a reduction path to the final answer of the program. Thus all reduction strategies that compute a normal form will compute the same normal form. However, not all strategies arrive at a normal form.

Most sequential implementations of reduction use one of the following two reduction strategies: *normal order* reduction and *applicative order* reduction. The normal order strategy takes the leftmost outermost redex as the next expression to be reduced. While the program has not yet reached its normal form, the leftmost outermost redex is always uniquely defined. Therefore normal order reduction implies sequential evaluation. The applicative order strategy takes one of the innermost redexes as the next expression(s) to be reduced, for instance the leftmost innermost one when sequential evaluation is aimed for. Some practical strategies for parallel reduction are discussed in section 3.3.

Normal order reduction has better termination properties than applicative order reduction. The leftmost outermost redex is always needed to construct the answer of the program. In contrast, an innermost redex may not be needed in the final answer (for instance when it is part of a conditional expression). An applicative order reduction strategy risks to get stuck, evaluating a non terminating expression that is not needed for the answer of the program.

As an example of reduction rules we will briefly illustrate the α- and β–reductions of the lambda calculus. Consider for instance the following reduction step:

$$(\lambda x\ .\ x\ y\ x)\ a \quad \rightarrow \quad a\ y\ a$$

In the lambda calculus terms are built by juxtaposition of variables (like *x*, *y* and *a*) or other terms. The juxtaposition represents function application. A term built from two variables like *x y* means the application of the function *x* to the argument *y*. A term like *x y x* should be interpreted as *(x y) x* : the function *(x y)* applied to the argument *x*: Thus juxtaposition associates to the left.

Prefixing an expression by a lambda followed by a variable and a dot, like in $\lambda x\ .\ x\ y\ x$, is called abstraction. It means that a function is created, where the variable x will later be replaced by the argument that will be given to this function. Thus applying $\lambda x\ .\ x\ y\ x$ to a variable *a* yields *a y a.* This substitution is called β-reduction. The variable *x* is called *bound* by the lambda prefix. The variable *y* is called a *free* variable in the term $\lambda x\ .\ x\ y\ x$, because it is not bound by any lambda.

Although the substitution mechanism implied by β-reduction may seem rather simple, there is a subtle difficulty that is illustrated by the next example, where two consecutive β-reduction steps are executed incorrectly:

$$(\lambda x\ .\ (\lambda y\ .\ x\ y\ x))\ y\ b \quad \rightarrow \quad (\lambda y\ .\ y\ y\ y)\ b \quad \rightarrow \quad b\ b\ b$$

In the first step the free variable *y* becomes bound by accident, just because it is substituted in a lambda term that happens to be an abstraction of *y*. Renaming the inner lambda abstraction (for instance *y* to *z*) prevents the accidental binding, without changing the meaning of the expression:

$$\begin{aligned}(\lambda x\ .\ (\lambda y\ .\ x\ y\ x))\ y\ b \quad &\rightarrow \quad (\lambda x\ .\ (\lambda z\ .\ x\ z\ x))\ y\ b \\ &\rightarrow \quad (\lambda z\ .\ y\ z\ y)\ b \quad \rightarrow \quad y\ b\ y\end{aligned}$$

Before β-reduction is performed the term $\lambda y\ .\ x\ y\ x$ is first replaced by $\lambda z\ .\ x\ z\ x$, which prevents the undesired binding that would otherwise arise during the first substitution. This renaming is called α-reduction.

The substitution mechanism implicated by β- and α-reduction is probably too complicated to serve as a primitive machine model [BER75, KLU83]. It can be shown that under certain conditions α-reduction can be omitted for sequential reduction [PEY87a]. However, if parallel reduction has to be possible, both the α- and the β-reduction must be implemented.

2.3 Implementation techniques

The implementation of reduction has already a long history. In 1960 the language LISP was proposed by McCarthy, as an implementation of the lambda calculus (without α-reduction!). In 1963 Landin published an abstract machine called the SECD machine, and presented a compilation scheme for LISP to this machine. The abbreviation stands for Stack, Environment, Code and Dump. In the next section the environment model will be discussed as one of the possible implementations of β-reduction.

In 1971 Wadsworth proposed to implement the lambda calculus by the use of graph-reduction [WAD71]. The graph structure allows shared sub-expressions and saves computation time when the shared expression contains redexes. Turner combined in 1979 graph-reduction with combinatorial logic to implement a functional language of his own design, named SASL (St. Andrews Static Language). He was the first who circumvented the difficulties in implementing β-reduction by the use of combinators [TUR79]. The next improvement in the implementation technique of functional languages was to replace the fixed combinators used by Turner with general program-derived combinators. This invention due to Hughes [HUG82] was almost immediately followed by the idea to implement program-derived combinators by compilation to an abstract machine. Compiled graph-reduction was first implemented by Johnsson and Augustson [JOH84], using an abstract machine that was similar to the SECD machine of Landin.

Before discussing the implementation techniques of parallel reduction we will first describe the essentials of the implementation methods that have been mentioned in the historical outline above.

2.4.1 Basic reduction mechanisms

Three major techniques exist to implement the substitution mechanism that is fundamental for all reduction systems. They are called *string reduction*, *graph reduction* and *environment reduction*. The β-reduction rule of the lambda calculus will be used in this section to illustrate the difference between the three methods. The techniques differ in the way the substitution process is implemented. String reduction performs substitutions literally, duplicating expressions (and work!) if needed. Graph reduction avoids the duplication of expressions by substituting only pointers to expressions. Finally, environment reduction performs no substitutions at all, but adds to each expression a table that associates the names of bound variables, occurring in that expression with the values to which the variables are bound. To illustrate the difference between the three substitution methods more clearly we show the reduction of the following example for each method:

(λx . x y x) a → a y a

2.4.1.1 String reduction

The way in which the reduction example is presented already illustrates the technique of string reduction. In the reduction step the variable *x* is bound to the expression *a*. Next, each occurrence of *x* is replaced by the literal text (*string*[1]) of *a*. The example also shows that the string *a* is duplicated by the substitution process.

To compare string reduction with the other two techniques figure 1 shows a graphical illustration of the same substitution. This graphical representation of lambda terms shows explicitly the presence of function applications in the form of *apply* nodes (indicated by an @). Remember that a space in for instance *(x y)* means: apply *x* to *y*. Figure 1 also shows an explicit node for each lambda abstraction.

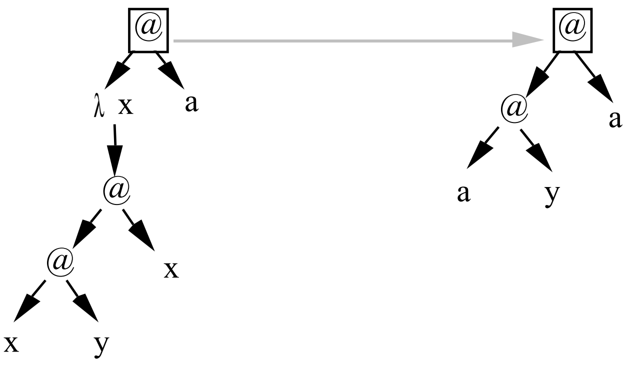


Figure 1: String reduction of *(λx . x y x) a → a y a*

The dashed arrow in figure 1 indicates the rewriting of the tree structure on the left-hand side to a new tree structure on the right-hand side. The top nodes of both trees have been drawn within a box to indicate that they are both the *same* physical node, representing the value of the expression. All other nodes on the right-hand side of the arrow (included the nodes in the subtrees *a*, *y* and *a*) are *new*. In particular the subtree *a* has been copied twice. This duplication of expressions during the substitution process is the reason that string reduction systems have a poor performance. Especially in recursive functions duplication can give rise to an exponential growth of work. In reduction systems that use an applicative order strategy, the pain of duplication is relieved somewhat. The expressions to be duplicated are first reduced to normal form (because they contain innermost redexes) and then duplicated. The normal form of an expression contains no more work, and thus only data is duplicated. In chapter 5 special attention is paid to the issue of avoiding duplication of work in our parallel reduction model that is partly based on string reduction.

[1] In this context strings are always structured objects in which e.g. brackets indicate the structure.

2.4.1.2 Graph reduction

Figure 2 illustrates graph-reduction of λ-terms as proposed by Wadsworth [WAD71]. In the reduction step of figure 2 only *one* new node is created to build the graph on the right hand side of the dashed arrow. This new node is the apply node below the top node. All other nodes are *shared* with the graph on the left hand side of the arrow (Alhough the top nodes of both graphs have been drawn seperately, they are the same physical node, indicated by boxes as before).

The algorithm proposed by Wadsworth tries to share as much of the original graph as possible. In figure 2 this is illustrated by the sharing of node *y*. On the right hand side a pointer to node *y* is used instead of creating a new instance of the whole subgraph rooted at *y*, as would happen with string reduction.

If sharing is already present in a graph it is maintained by the reduction algorithm. In figure 2, sharing of node *x* on the left hand side results in sharing of node *a* on the right hand side. The substitution of *a* into *x*, is mechanically performed by storing twice a pointer to *a* in both apply-nodes of the right hand side graph.

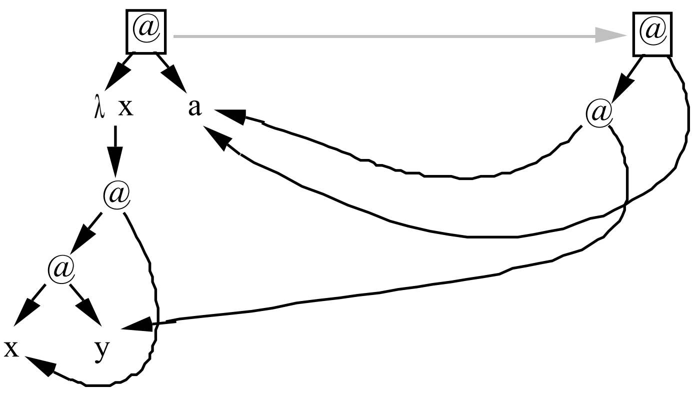


Figure 2: Graph reduction of *(λx . x y x) a* → *a y a*

It should be noted that the graph on the left hand side of the dashed arrow can not be discarded after the reduction step (i.e. "garbage collected"). As graph reduction creates sharing of nodes (node *y* in the example), any node of the left hand side graph may already be shared before the reduction step begins. To recover storage space occupied by those parts of the program graph that are no more referenced a *garbage collection* algorithm has to be used.

The sharing of bound variables in a lambda expression (*x* in the example) and the preservation of this sharing by the graph reduction algorithm, is the reason that work will never be duplicated, as was the case in string reduction. Graph-reduction can therefore be performed efficiently, even when reduction is performed in normal order.

2.4.1.3 Environment reduction

Environment reduction is used in most LISP-interpreters. The *environment* is a special data structure in which variable names are associated with their values. On the left hand side of figure 3 the application node below the lambda node contains such an environment, in which the variable *y* is bound to the value *2.* This binding has been established by a former reduction step. The current reduction step that is illustrated in figure 3 copies the application node below the lambda node into the root node, extending the environment with the information $x = a$.

In contrast to graph reduction (and string reduction), environment reduction performs no substitutions when a lambda expression is applied. Instead the environment is extended with the value of the variable that is bound. In figure 3, the right hand side graph is constructed without creating any new nodes.

Although it seems that no substitution is performed, the environment has to be dereferenced when the value of a variable is needed. The reduction process as illustrated in figure 3, is only part of an environment based reduction system. The way in which the environment is built and dereferenced forms a major part of the (in)efficiency of such a system.

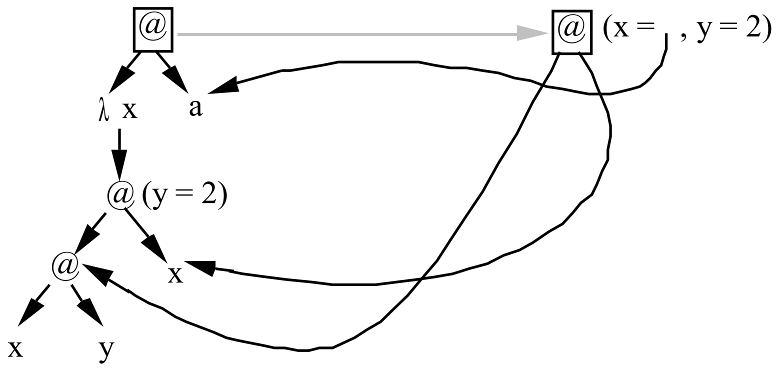


Figure 3: Environment reduction of $(\lambda x\, .\, x\ \ y\ \ x)\ \ a\ \rightarrow\ \ \ \ a\ \ y\ \ a$

A mechanism to avoid the use of names in the environment has been proposed by De Bruijn [BRUI72]. The variables in an expression are replaced by a number that indicates the lambda to which the variable belongs. This number is equal to the number of lambda symbols encountered on the way up in the graph, form the variable up to (but not including) the lambda symbol to which it belongs. For instance $\lambda x\, .\, (x\ (\lambda y\, .\, x\, y))$ becomes $\lambda\, .\, (0\ (\lambda\, .\, 1\ 0))$. The lambda symbols become anonymous, and the variables are a kind of relative pointers to the lambda symbols. Therefore it seems as if α-reduction is no longer needed. In stead, a renumbering of variables can be performed in substituted terms. However, this is computationally just as expensive as α-reduction.

De Bruijn numbers can be considered as an offset in the environment, when the value of a variable is required. This property is for instance used by Curien in his categorial abstract machine [COU85].

The implementation of functional languages by a combination of De Bruijn numbering with environment reduction and an eager reduction strategy is very similar to the implementation of an imperative language like PASCAL on a stack based abstract machine. The environment that is created by the lambda applications is equivalent to the stack-frame that is created by an imperative function call. In both cases function arguments are first evaluated before being added to the environment. Variables in an expression are replaced by offsets in the run-time created environment. It is not surprising that compilers for eager functional languages generate code with the same efficiency as is usual for imperative languages.

2.4.2 Combinator reduction

In its full generality the process of β-reduction appears to be difficult to implement. However, more elementary substitution rules can be used to achieve the same effect as β-reduction. In 1924 Schönfinkel [SCHÖ24] presented a calculus of functions (in which he also introduced higher order functions and partial application, later unfortunately called Currying) based on a set of three elementary functions (verSchmeltzungs-, Konstanz-, Unverträglichkeitsfunktion: *S*, *C* and *U*). He showed that any formula in first order predicate calculus could be transformed into an expression merely consisting of applcations of these elementary functions. Following this idea, it can be proved that expressions based on a fixed set of simple lambda-terms (e.g. only the *S* and *C* of Schönfinkel) have the same computational power as the lambda-calculus. These simple lambda terms have been called *combinators*, because they only specify combinations of their arguments.

Barendregt [BAR84] defines combinators as lambda terms that do not contain free variables (such terms are also called *closed terms*). However, combinators used in practice are more restricted. To simplify mechanical interpretation of combinators by rewrite rules, lambda abstractions inside a combinator body are not allowed. Thus, a practical combinator corresponds to a closed lambda expression, in which all lambda abstractions occur directly concatenated to left side of the expression, like: $\lambda x.\ (\lambda y.\ (\lambda z.\ \ldots\ ))$.

The *S*, *K* and *I* combinators proposed by Schönfinkel (apparently, the *C* combinator has later been renamed to *K*) are equivalent to the following lambda terms:

$$S = \lambda x\,.\,(\lambda y\,.\,(\lambda z\,.\,x\ z\ (y\ z)))$$
$$K = \lambda x\,.\,(\lambda y\,.\,x)$$
$$I = \lambda x\,.\,x$$

From the point of view of mechanical interpretation, combinators can also be considered as elementary rewrite instructions of an abstract reduction machine. To illustrate this view, we show the behaviour of the *S* combinator as implied by the definition of *S* as a lambda term. An application of the *S* combinator to three arguments (*a, b* and *c*) is replaced by the equivalent lambda expression:

$$S\ a\ b\ c \quad = \quad \lambda x\,.\,(\lambda y\,.\,(\lambda z\,.\,x\ z\ (y\ z)))\ a\ b\ c$$

The lambda expression can be reduced, applying three β-reductions (multiple β-reductions are indicated by $\rightarrow^*$):

$$\lambda x\,.\,(\lambda y\,.\,(\lambda z\,.\,x\ z\ (y\ z)))\ a\ b\ c \rightarrow^* \quad a\ c\ (b\ c)$$

When we conceive the operation of the *S* combinator as a single action instead of three β-reductions, we obtain the view of a *rewrite* mechanism. The reduction of (*S a b c*) is now defined as a single *rewrite rule* with the following pattern:

```
S a b c    →    a c (b c)
```

A rewrite rule specifies how an expression is textually rewritten to another expression by giving two *patterns* connected by an arrow. The left-hand side pattern consists of the name of the rewrite rule (*S*) plus a number of variables (*a,b,c*) that have to match a given expression. For instance the expression: *S (K u v) w (S u)* has to be rewritten in the following way:

$$\underline{S}\ \underline{(K\ u\ v)}\ \underline{w}\ \underline{(S\ u)} \quad \rightarrow \quad (\underline{K\ u\ v}\ (\underline{S\ u}))\ (\underline{w}\ (\underline{S\ u}))$$

$$S \quad a \quad b \quad c \quad \rightarrow \quad a \quad c \quad (b \quad c\ )$$

left hand side rewrites to right hand side

The variables *a*, *b*, and *c* on the left hand side of the *S*-rule can match arbitrary expressions. In the example *a* matches *(K u v)*, *b* matches *w* and *c* matches *(S u)*. The right hand side of the *S*-rule specifies the resulting expression, when the variables are replaced by the expressions to which they were matched.

Interpreted as rewrite rules the *S*, *K* and *I* combinators constitute the following rewrite system:

```
S a b c    →    a c (b c)
K a b      →    a
I a        →    a
```

This rewrite system is sufficiently simple to be considered as an abstract machine model that may be implemented on a concrete machine architecture.

All closed lambda terms can be translated to terms only containing the *S*, *K*, and *I* combinators. For example the expression used in the first section becomes:

$$(\lambda x\,.\,(\lambda y\,.\,x\ y\ x)) \quad = \quad S\ S\ K$$

An application of this lambda expression can be reduced according to the given rewrite rules for *S*, *K*, and *I* :

$$S\ S\ K\ a\ b \rightarrow \quad S\ a\ (K\ a)\ b \quad \rightarrow \quad a\ b\ (K\ a\ b) \rightarrow \quad a\ b\ a$$

Exactly the same result is obtained in four rewrite steps as would be obtained by performing the two β-reductions in the original lambda term. Because the rewrite actions described by the combinator rules are less complex than β-reduction, more rewrite steps are required.

In 1979 Turner [TUR79] published an efficient compilation scheme to translate the functional language SASL into a fixed set of a few dozen combinators, all very similar to the SKI combinators. Turner also proposed an abstract graph reduction machine based on rewrite rules associated with the combinators. Inspired by this technique, a number of machine architectures have been constructed or simulated (SKIM [CLAR80], NORMA [SCH86]). Also SKI-based parallel string reduction machines have been proposed (COBWEB [SHU85]).

2.4.3 Lambda lifting

It was soon realized that the granularity of the rewrite actions specified by the SKI-combinators is too fine to be implemented efficiently in hardware (sequential or parallel). It appears [HAR89] that each SKI-type reduction produces an intermediate result that is largely taken apart again in the next reduction step. The storage and reclamation of these intermediate results can be avoided if larger grain combinators could be devised.

Hughes proposed an algorithm to derive larger combinators from the source text of the user program [HUG82]. He called these program-derived combinators *super-combinators*. The rewrite behaviour of these combinators depends on the contents of the user-program. Therefore, the set of combinators has become variable (in contrast with the fixed SKI-set), which seems a disadvantage for the design of special architectures.

A practical algorithm to derive combinators from the user program has been proposed by Johnsson [JOH85], called lambda lifting. With the technique of lambda lifting, a functional program can be converted into a set of possibly recursive, coarse grain combinators. Consider the following definition of a function *F* in which *H* is a combinator that has been defined before (and is treated as a constant):

$$F = (\lambda y . H\ (\lambda x . y))$$

The body of *F* contains a lambda expression and is therefore not a practical combinator (*F* can not be considered as a rewrite rule because at the time of rewriting it is not known to which value x will be bound). The nested lambda expression $(\lambda x . y)$ is not a combinator either, because it contains a free variable *y*. The idea behind lambda-lifting is to convert all free variables into bound variables. In the example this is done by adding a lambda abstraction to bind *y* in the inner lambda term and applying the new abstraction to the variable *y*:

$$\text{replace} \quad (\lambda x . y) \quad \text{by} \quad (\lambda y . (\lambda x . y))\ y$$

Applying this conversion to the body of *F* yields the following definition:

$$F = (\lambda y . H\ (\lambda y . (\lambda x . y))\ y)$$

The meaning of the function *F* has not been changed, but now the sub-expression $(\lambda y . (\lambda x . y))$ fulfils the criteria of a combinator and can be *lifted* out of the definition of *F*. If we call the lifted expression *G*, the following result is obtained:

$$F' = (\lambda y . H\ G\ y)$$
$$G = (\lambda y . (\lambda x . y))$$

Also *F'* has become a valid combinator, because the body of *F'* contains the combinator *G* in stead of the original lambda expression. As before, combinator names (H and G inside F') are considered as constants.

The function *F* has been converted into a set of combinators: *F'*, *G*. These combinators can now be interpreted by an abstract rewrite machine, in the same way as the SKI combinators were interpreted in Turners machine. The grain size of the program-derived combinators is larger, but the set of combinators is not fixed any more, because the combinator definitions derived by λ-lifting depend on the contents of the user program.

The advantage of the lambda lifting transformation is that the resulting combinator definitions can be considered as locally confined, coarse grain rewrite actions. When the left hand side of a rewrite rule has been matched to a part of the main expression, the right hand side pattern together with the obtained bindings for the variables constitute a coarse grain of computation that may be well suited for distribution in a parallel architecture. The presence of free (global) variables would complicate parallel reduction, because the value of such a free variables may be determined by other (rewrite) processes. Synchronization and communication may be needed in the middle of a rewrite action, when the value of such a free variable is needed. Combinators do not have free variables (all variables of a combinator are bound before rewriting is performed) so a rewrite action can be completed without intervening communication.

The grain size of program-derived combinators as they are produced by lambda lifting, may be so large that a significant opportunity for parallel evaluation is lost. To increase the amount of available parallelism, Hudak proposes another transformation, which generates finer grain combinators that are guaranteed to have no parallel sub-structure [HUD85]. These combinators are called *serial-combinators.* In chapter 5 serial combinators are compared with the job-concept, which is proposed as an essential part of our parallel reduction model.

2.4.4 Term rewriting

Pattern matching is considered to be an important feature in functional languages. It increases the readability of functional programs. Functions are only capable to generate one result. In large programs function-results become very complicated structures. In general these structures have to be taken apart in order to provide input for other functions. Pattern matching is an elegant mechanism to specify in a function definition which parts of the input structure are going to be used.

Although pattern matching can be compiled to nested conditional expressions, it can also be considered as an essential part of the computational mechanism. The inclusion of pattern matching in rewrite rules results in a so called term rewrite system. The rules are called *Term Rewrite Rules*, because both the left-hand side and the right-hand side of the rules may be general terms. Consider for example the definition of a rule that reverses both elements of a *Cons*-pair:

Reverse (Cons x y) → (Cons y x)

The expression looks like a combinator definition, but the argument of *Reverse* is not a variable, but a term: *Cons x y*. When an expression is rewritten according to the rule for *Reverse*, the argument to *Reverse* has to be matched against the pattern *Cons x y*. In this pattern the variables *x* and *y* match any expression, but the function identifier *Cons* only matches *Cons*:

$$\begin{array}{lllllllll}
\text{Reverse} & \text{(Cons} & \text{(}\underline{\text{Cons 3 4}}\text{)} & \underline{\text{5}}\text{)} & \rightarrow & \text{(Cons} & \underline{\text{5}} & \text{(}\underline{\text{Cons 3 4}}\text{))} \\
\textit{Reverse} & \textit{(Cons} & \textit{x} & \textit{y)} & \rightarrow & \textit{(Cons} & \textit{y} & \textit{x \quad)} \\
\text{left hand side} & & & & \text{rewrites to} & & \text{right hand side} &
\end{array}$$

During the pattern match the variable *x* is bound to the term *Cons 3 4*, and the variable *y* is bound to *5*.

Because a pattern might fail to produce a match, a set of alternative rules with the same name but different patterns is allowed in a term rewrite system. With such a set of alternative rules often complicated nested conditional expressions can be avoided. In general, the use of rules with alternative patterns considerably reduces the amount of conditional expressions compared to pure combinator systems. However, in terms of performance it remains questionable if the advantage of less conditional expressions outweighs the increased complexity of the computational mechanism by the pattern match algorithm (compared to pure combinator rewriting).

Combinators can be considered as a special case of term rewrite rules. A combinator is a term rewrite rule, where the arguments of the rule are only allowed to be variables instead of general terms. Within the context of the Dutch Parallel Machine Project the research group in Nijmegen has pursued the approach of using term rewriting as the basic model of computation [BAR87]. For sequential machines they have been able to show that is is possible to implement term rewriting efficiently via graph reduction.

2.4.5 Compiled reduction of combinators

A fixed set of combinators can be implemented by manually translating the rewrite action corresponding to each combinator into the machine language of a given architecture. This results in a fixed set of machine language functions, one for each combinator. This approach is not practical for program-derived combinators because the manual translation would have to be repeated for each new program. A mechanical translation scheme has to be found to map program-derived combinators onto a given architecture. Several solutions to this problem have been described in literature [JOH84, FAI87, BRU87]. Some of these proposals introduce an abstract machine that is oriented towards graph reduction. The abstract graph reduction machine designed by Johnsson, called the *G*-machine (Gøthenborg-machine), has been seriously considered as a candidate for a special (sequential) architecture. The proposal by Fairbairn is very close to a conventional stack machine.

Chapter III

INTRODUCTION - Architectures for parallel reduction

3 Architectures for parallel reduction

The implementation techniques discussed in the previous chapter, can be used to map the reduction model of computation onto a concrete computer architecture. To compare such mappings for several reduction machine designs, we introduce two models to describe the hardware- and software architecture of (parallel) machines.

In addition to the implementation techniques for reduction, these two models introduce architectural concepts that we need to construct a uniform framework for the comparison of parallel reduction architectures. The goal of this qualitative comparison is to identify the essential aspects in which our own reduction architecture differs from related machines. In sections 3.1, 3.2 and 3.3 an architectural reference framework is established. In Section 3.4 several parallel reduction machines (including our own architecture) are compared with reference to the framework. Finally, section 3.5 presents some design considerations for the architecture of our reduction machine. The discussion in chapter 3.5 supplements other design considerations in chapter 4.

3.1 Architecture models

Computer architectures can be considered from at least two different viewpoints. An architecture can not only be seen as a layered structure of hardware components, but may also be viewed as a hierarchy of abstract machines.

The first view is reflected in the hardware architecture model of figure 1. This model describes the structure of physical objects that make up a computer. When a number of physical components perform a certain distinct function, this group of objects will appear as a basic element in the next higher layer of the model.

The second view gives rise to the software architecture model of figure 1. This model describes the structure of the instuction code that is stored in the computer. When a piece of code performs a certain distinct function, it becomes a basic instruction in the next higher layer of the model.

hardware architecture model	software architecture model
	functional language layer
	reduction machine layer
Program Memory Switch layer	abstract machine layer
Register Transfer layer	machine language layer
	micro-programming layer
logic layer	
circuit layer	
.....	

Figure 1: A hardware and software architecture reference model.

Most proposals for parallel reduction machines do not pay much attention to architectural models. Consequently there is no universal agreement, neither on terminology nor on models for architectures.

We adopt a model from Bell and Newell [BEL71, BEL79] for the hardware architecture. The model used for the software architecture is similar to the one described by Tanenbaum [TAN84], but in this context the layers and terminology are specially adapted to reduction machines. A relation between the two models is suggested in figure 1, by drawing corresponding layers of both models at the same height. The relation will be explained during the description of the software architecture model. First, we shortly describe the hardware architecture model.

On the lowest level of the hardware model, called the *circuit* layer, a computer consists of passive and active electrical circuits. The behaviour of the machine is described in voltages and currents. On the next higher level, the *logical* layer, an architecture is represented in terms of logical circuits. The behaviour of these circuits can be described by discrete states, like "true" and "false". The representation on the logical level uses no information typically belonging to the circuit layer, like current or voltage. In other words: the logical layer abstracts away from electrical characteristics.

In the third layer a computer is viewed as a collection of registers and arithmetic functions operating on the contents of these registers. The notion of boolean values of the previous layer is now replaced by the concept of binary numbers. The layer is called the *Register Transfer* layer (RT-layer). The description of an architecture on this level is a collection of rules that specify the transfer of numerical values between registers and functional units. The rules are triggered by conditional expressions on the state of the machine.

On the highest level a hardware architecture can be considered as a collection of processors, memories, communication switches, input-output controllers etc. The layer corresponding to this level has been called *PMS* -layer, which is an abbreviation of the three most important components on this level: Processors, Memories, Switches. The PMS-layer presents the overall structure of a computer in such a way that performance aspects can be easily identified. Typical parameters of this layer are the rate of data transfers, the size of register banks, caches, memories, the capacity of processors etc.

It is interesting to note that several years after Bell and Newell proposed the model, functional units on the RT-level became available as LSI-circuits. Nowadays, even modules on the PMS-layer are integrated into VLSI designs. In future on-going miniturization will give rise to the integration of several PMS-functions into one module. The GAP-chip [DAV84], which contains 72 simple processors connected into an array structure, is an early example of this trend. May be such a development leads to the necessity to introduce a layer above the PMS-layer, in which complete sub-architectures are distinguished as basic components.

All layers of the hardware model always represent a parallel machine. Each layer describes the behaviour of the architecture with respect to time. At any instant of time many actions may take place in parallel. The concept of a sequential machine only exists in the software architecture.

The second view of a computer architecture, has been proposed by Tanenbaum [TAN84], although it is claimed by Bell, Mudge and McNamara [BEL79] that the idea is due to J.V. Levy (1974). In this view the software architecture of a computer is considered as a hierarchy of interpreters (or compilers) of abstract machines.

The lowest layer of the software model describes a computer as a collection of basic sequential machine instructions. Therefore it is called the *machine language* layer. The instructions are interpreted by a piece of hardware that is specified on the RT-level. That is why the machine language layer in figure 1 is drawn on the same height as the RT- layer of the hardware model. The illustration stresses the fact that the machine language layer corresponds to the RT-layer.

Tanenbaum [TAN84] distinguishes a layer below the machine language layer, called the micro-programming layer. However, this layer is not applicable to all architectures. Micro-programming can be viewed as an implementation of the RT-level specifications of the hardware model. Other (and faster) implementations exist on this level to achieve the same result. In modern RISC designs the micro-programming layer has been abandoned in favour of a lower level (sequential) machine language layer.

The next layer of the abstract machine architecture is called the *operating system* layer [TAN84]. In conventional machine architectures this layer is implemented by a machine language program, called the operating system. In experimental architectures only certain functions of a conventional operating system are actually implemented. The abstract machine model that is provided on this level may be quite different from the machine model offered by conventional operating systems. Therefore we prefer to call this layer the *abstract machine* layer.

The abstract machine can be considered as a programming model corresponding to the PMS-layer of the hardware model. The machine language layer is extended by instructions to support, for instance, process management, memory management, communication and load distribution. These activities correspond to components that are distinguished in the PMS-layer of the hardware architecture. Therefore the abstract machine layer in figure 1 is drawn at the same height as the PMS-layer.

The third layer of the software model is called the *reduction machine* layer. On this level the basic reduction mechanism is supported by a suitable abstract instruction set (e.g. combinators, G-code). The instruction set may be compiled to instructions of the abstract machine layer. Older proposals for reduction architectures [TUR79] use an interpreter to implement operations of the reduction machine layer.

Special hardware reduction architectures like the G-machine [KIE85], SKIM [CLAR80] and NORMA [SCH86] provide an instruction set on the machine language level that is specially tailored to support operations of the reduction machine layer. This optimization is primarily reflected in the RT-layer of the architecture and in some respects also in the PMS-layer, e.g. a parallel garbage collection processor.

The highest layer of the software architecture model is called the *functional language* layer. Although many functional languages exist, they do not give rise to differences in the reduction machine layer that are significant from the architectural point of view. After compilation to the reduction machine layer, most aspects of a functional language that are important for the lower layers of the implementation are still manifest (e.g. normal order semantics, user-annotations).

The implementation techniques for reduction as discussed in the previous chapter (e.g. translation to program-derived combinators) are equally applicable to all functional languages. The complexity to derive information on sharing of expressions and strictness of functions is the same for all functional languages.

Some functional languages (e.g. SASL [TUR79]) allow less restricted types than other languages (e.g. Miranda [TUR85]). In the implementation this may lead to run-time checks on the type of expressions. These checks can be omitted in strongly typed languages resulting in higher reduction speed. For the present discussion of parallel reduction architectures we ignore this aspect and consequently differences in functional languages do not need to be considered.

3.2 Lower levels of the framework

The reference framework that will be established consists of a set of architectural properties. These properties represent essential aspects of the implementation of parallel reduction, corresponding to all layers of the hardware and software architecture model. The higher level properties in the framework are more reduction specific, whereas the lower level properties are related to parallel architectures in general. Therefore we split the presentation of the framework and the comparison in two parts.

The first part is described in this section and covers the architectural properties of the abstract machine layer (software model) and the corresponding PMS-layer (hardware model). Relevant properties of lower layers are also included in the first part.

The second part that will be presented in the next section covers the higher layers of the software model.

The discussion of the abstract machine layer in literature on parallel reduction machines is often rather superficial. The abstract machine is frequently presented by a model that describes the interaction between processes and storage (a process-storage model [WAT87]). Such a model gives a global impression of the abstract instructions concerning process management, memory management and communication. For a number of parallel reduction machines some abstract machine models and their corresponding PMS-layer descriptions will be reviewed in the next subsections.

3.2.1 Alice, Flagship and Rediflow

An abstract machine model that is used in several parallel reduction machine designs (Alice, Flagship, Rediflow) is shown in figure 2.

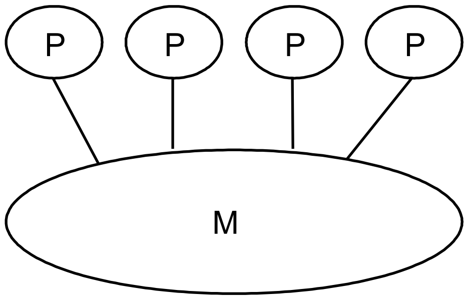


Figure 2: The abstract machine model of the Flagship- and Rediflow architecture.

In this model a number of processes (P) can access concurrently one shared storage space (M). Processes may be dynamicly created and deleted and communication may occur between each pair of processes. The abstract machine model of figure 2 is similar to the abstract machine model found in conventional sequential architectures. It does not reflect the differences in communication- and memory bandwidth that may exist in the hardware.

The architecture of figure 2 may be implemented by different PMS-descriptions of the hardware. Three configurations are shown in figures 3a, 3b and figure 4.

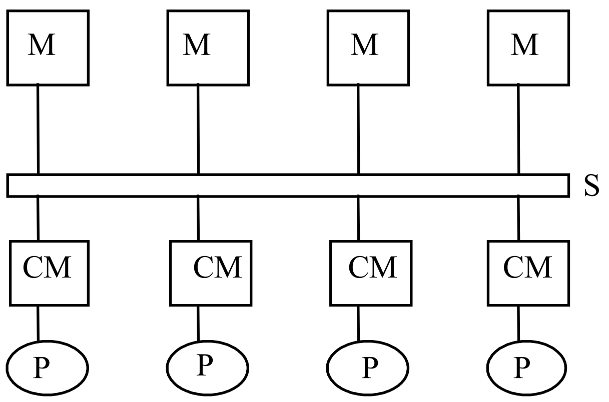


Figure 3a: The PMS-description of a cache-based shared memory architecture.

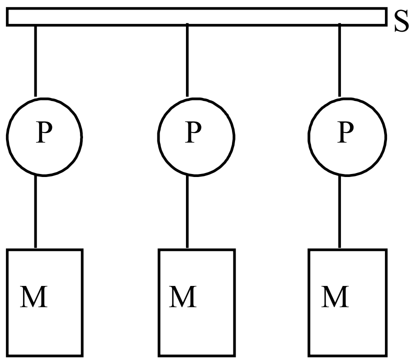


Figure 3b: The PMS-description of the Flagship architecture

In figure 3a each processor (P) has direct access to a cache memory (CM) and shared access (via the cache) to a partitioned global memory (M). The communication switch (S) may serve several memory accesses simultaneously. In this way partitioning of the shared memory can be exploited. In addition the caches will avoid many accesses through the switch, if computations exhibit locality.

The architecture of figure 3a is currently used in several experimental parallel machines [PFI85, NGU88, WIL88]. The implementation of reduction on such an architecture is particularly interesting because the property of referential transparency may reduce the cost of maintaining cache coherency. For instance, when expressions are cached that are in normal form, they will never be updated. On the other hand, if a redex happens to be copied in several caches, all those copies will eventually reduce to the same value. When, after reduction of a cached redex, coherence between caches and main memory is not reestablished, only computation time will be wasted, incorrect answers will not be produced. Therefore, (cheaper) variants of coherence mechanisms may be possible that do not maintain complete coherence at any instant of time.

An architecture similar to the one in figure 3a that is used for parallel reduction is the Alice machine [DAR81, EIS87]. However, no hardware supported caches are provided. Instead reduction processors are provided with small local memories where function definitions are stored in a cache-like manner. To access and rewrite redexes, procesors always have to access main memory through the communication switch. In the Alice machine, processors are clusters of Transputers [WHI85] and the communication switch is implemented as a multi-stage delta-network based on a specially designed ECL-chip.

A different approach is taken by the Flagship project [WAT86, WAT87]. In comparison to the architecture of figure 3a, one could say that the cache size is significantly increased and the main memory is completely discarded. However, the memories in the Flagship machine are no hardware caches but normal memories. Each local memory contains part of the global address space. These parts may overlap and copies of the same subgraph may exist in several local memories, which bears some ressemblance to caching. It is not yet specified how coherence is dealt with. The data communication switch will be probably implemented as a delta-network.

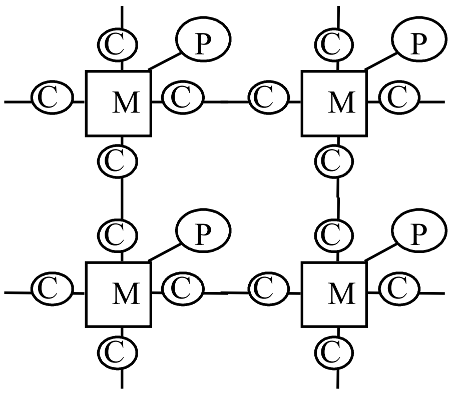


Figure 4: The PMS-description of the Rediflow architecture.

Figure 4 shows the PMS-level architecture that has been used by Keller in a simulation of the Rediflow machine [KEL84, KEL86]. A number of identical processing elements are configured into a regular 2-dimensional mesh-structure. Each processing element consists of a memory connected to five processors. Four of these (C) are specially devoted to communication. They provide parallel DMA-transfer capability to the surrounding processing elements. The fifth processor (P) is a general purpose programmable processor used to implement the abstract machine. The processing element conceived by Keller is similar to the Transputer architecture [WHI85].

3.2.2 ZAPP

An abstract machine model that does not support a single addressable storage space is shown in figure 5. This model has been used for the implementation of parallel reduction in the ZAPP proposal [MCB87].

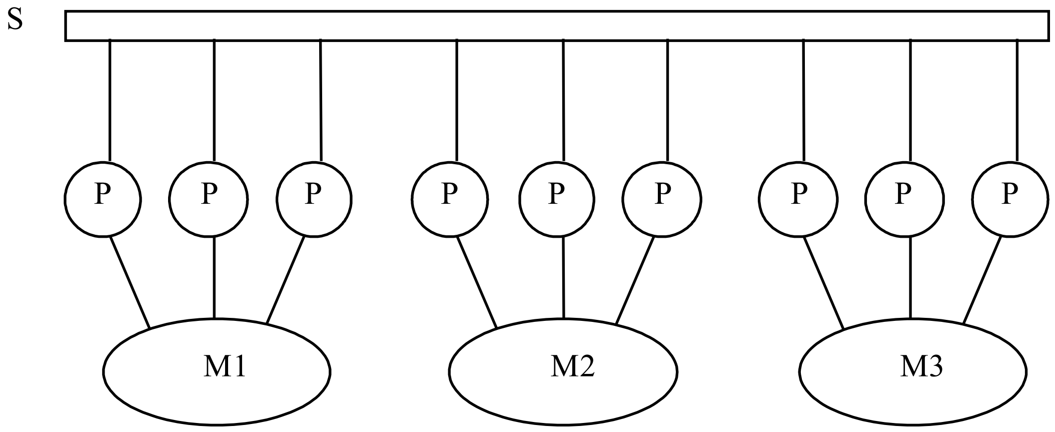


Figure 5: The abstract machine model of the ZAPP architecture.

The total storage space is divided in several disjunct fixed spaces (M). Each sub-space can be accessed by a number of processes. These processes may be created dynamicly but once in existence they are not allowed to migrate between memory spaces. Communication instructions are provided in the abstract machine to synchronize two processes and to transport data.

The ZAPP abstract machine model has been implemented on a PMS-level architecture identical to the one illustrated in figure 4. The main difference between the models of ZAPP and Rediflow is that the former model does not try to hide the relatively slow access to remote memories, while the latter architecture does.

3.2.3 GRIP

Although on a higher level the abstract machine model of the GRIP machine [PEY87b, CLA86] may be described by the model of figure 2, there is a fundamental asymmetry in the design that justifies a lower level illustration. The philosophy behind GRIP is to increase the grain size of the operations on the shared memory. Instead of simple read/write actions on a word by word basis, the memory (M) is provided with processing capacity to perform higher level (reduction specific) instructions.

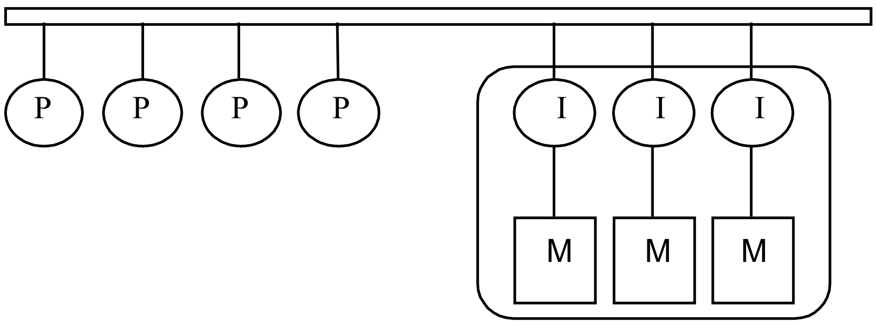


Figure 6: The abstract machine model of the GRIP architecture.

This results is an abstract machine model shown in figure 6. There is a pool of processes (P), each of which operates on one shared memory. The memory is accessed via intermediate special purpose processes (I). These processes perform reduction specific operations. A single global memory address space is realized by the global communication address space between processes *P* and *I*.

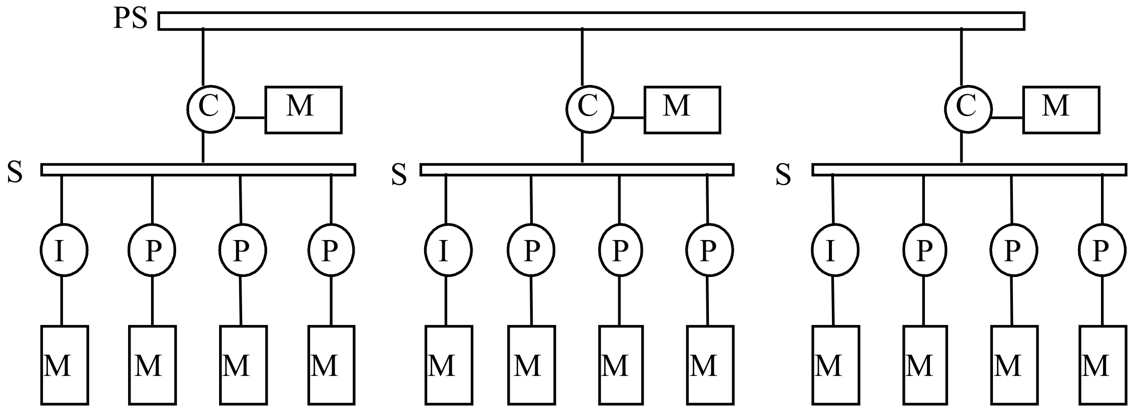


Figure 7: The PMS-description of the GRIP architecture.

Figure 7 illustrates the PMS-level architecture of the GRIP machine. Processors are clustered in groups. Each processor is provided with a local memory. There is one shared bus-connection (S) between the processors in a group. The interconnection between clusters is realized by a second shared bus-connection (PS) and a special interface processor (C). The processors in a cluster are not identical. One of them (I) performs specialized functions supporting graph reduction on a large local memory. The other four processors (P) only have a small local memory and are responsible for the actual graph reduction.

The specialized processors (I) are intended to increase the functionality of memory operations (intelligent memories). This may reduce the amount of memory operations while increasing the size of the information to be transported. Therefore it becomes possible to exploit a packet switch protocol on the bus *PS*. The local buses *S* are normal circuit switched (one word at a time) buses. It is expected that the packet switched bus achieves a high utilization factor, which would reduce the disadvantage of this bus as a potential bottle-neck in the architecture.

3.2.4 APERM

The abstract machine model of APREM (Amsterdam Parallel Experimental Reduction Machine) is similar to the ZAPP model. However, for efficiency reasons two abstract communication mechanisms are introduced.

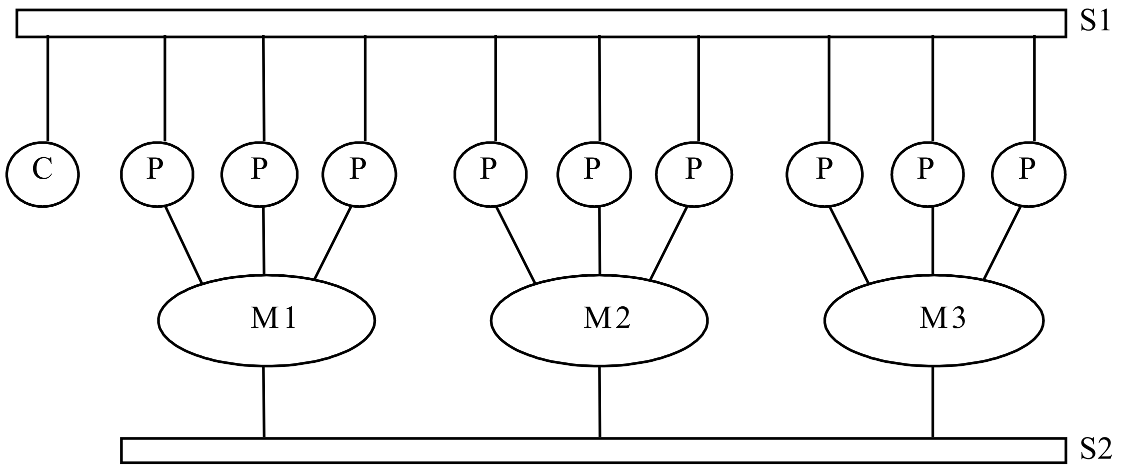


Figure 8: The abstract machine model of the APERM architecture.

The first mechanism allows processes to synchronize and transport a small amount of data (connection S1 in figure 8). The second mechanism copies large graph structures from one storage space to another. The copy is performed without process synchronization (connection S2 in figure 8). Synchronized transport involves a number of duplications of the data to be communicated. Some of these duplications can be avoided when data is transferred without process synchronization. A more detailed account on this issue can be found in chapter 8. As far as we know, our architecture is the first reduction machine that features a separation of synchronized process communication and unsynchronized bulk-data transport.

A second difference with the ZAPP abstract machine model is that processes in APERM are static. A fixed number of processes is associated to each storage sub-space.

In contrast to the other architectures, the current design APERM features a single process (C) to control the distribution of reduction tasks in the machine. The principles of loadbalancing mechanisms are compared in the next sections on the higher level aspects of architectures.

The PMS-description of APERM is illustrated in figure 9. It is composed of processors interconnected by dual ported memories. The advantage of such connections compared to a general communication switch is the high bandwidth. Data may be transferred from one memory to an adjacent memory at the full rate of one word per memory cycle. A more detailed account on the PMS-level design of APERM is given in section 3.5 and chapter 4. A more detailed account on the APERM abstract machine model is presented in chapter 8.

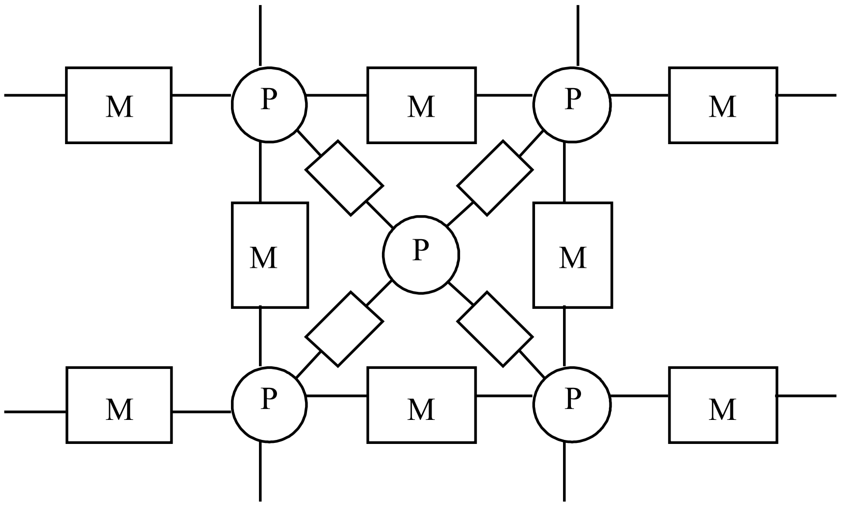


Figure 9: The PMS-description of the APERM architecture.

3.3 Higher levels of the framework

The higher layers of the abstract architecture constitute the second part of the reference framework. It mainly concerns the reduction machine layer and the translation to the abstract machine layer. The functional language layer is not considered because the differences between the various functional languages in use are not of significant importance from the architectural point of view (see section 3.1).

The following architectural properties (a-e) constitute the higher level part of the reference framework:

a) The representation of the program in the abstract reduction machine.

Possible representations are based on graphs, trees or strings. A tree representation can be considered as a graph without sharing. A string representation is a tree without pointers.

Strings are stored in consecutive storage places in the abstract reduction machine. Strings have to be delimited with special marks to indicate the beginning and the ending.

b) The reduction mechanism.

The instruction set of the abstract reduction machine has to support one of the basic reduction mechanisms: β-reduction, fixed combinator reduction or program-derived combinator reduction. The reduction machine is mapped onto the abstract machine by means of compilation or interpretation.

The reduction process on the level of the abstract reduction machine can be divided in three different activities:

1) finding reducible expressions
2) reducing these expressions
3) collecting garbage

Especially the first and the last activity probably consume a considerable amount of the computation power involved in reduction. Hartel discusses this issue in detail in his thesis on storage management in fixed combinator reduction machines [HAR89]. In the present comparison we will only briefly indicate what kind of garbage collection is proposed for the considered architectures:

c) Garbage collection method

d) Parallelism in the abstract reduction machine (reduction strategy)

The Church-Rosser property of practical reduction systems implies that multiple reducible expressions may be rewritten in parallel as long as it is guaranteed that the program terminates. Not all reducible expressions in a program are needed to compute the final answer of that program. The conditional function (if then else) is an example of a function that does not need all its arguments to compute its answer. A safe approach to parallel reduction is only to compute those redexes in parallel that are known to be needed for the final answer. This form of parallelism is called strict argument parallelism. A function is called *strict* in an argument if the argument is needed to compute the function.

The problem with strict argument parallelism is that it is not always possible to determine if an argument is needed in a computation. An additional problem is that only part of an argument may be needed. For instance one can imagine a computation that only needs all even elements of a given list. If such a list is computed in parallel to the main computation then twice the amount of work would be done compared to what is needed (assuming that all elements of the list require the same amount of computation). Still it may be advantageous to evaluate a "partly needed" data-structure in parallel, even though part of the work is not needed. On the other hand it might not be worth to perform a fully needed computation in parallel if its grain-size is too small.

Parallel reduction can be based on strict-argument parallelism or on other parallel strategies. An example of a safe parallel strategy that also reduces non-strict arguments is the Gross-Knuth strategy. A Gross-Knuth reduction step consists of two phases. In the first phase the set of all current reducible expressions is determined. During the second phase, all expressions in the set are rewritten in a random order. This means that the strategy repeatedly advances all computations by one reduction step. The random order of the reduction steps in the second phase implies that as much parallelism can be used as is available. Both needed and not-needed computations are performed and evenly spread over the available processing power. The Gross-Knuth strategy specifies fine grain parallel computations. However, grain-size can be increased by a variant of the strategy. In this variant all parallel computations are advanced by a fixed number of reduction steps instead of one. The Gross-Knuth strategy can be compared to a breadth-first evaluation of goals in a (parallel) inference machine.

All practical approaches to parallel reduction use strict argument parallelism. Methods have been developed to determine strictness information during compile time [PEY87a]. Strictness analysis may be supplemented with heuristic information about the grain-size of combinators [HUD85]. In such a way the triggering of parallelism is indicated by the compiler and the programmer does not need to provide knowledge about parallelism in the program.

Because in practice both strictness analysis and grain-size heuristics have not yet proved to be very effective, much better results can be obtained if some indications about these two properties are provided by the programmer.

In addition to the previous reduction specific properties (a,b and c) the following general issues concerning parallel architectures will also be considered in the architecture comparison:

e) Grain size

The grain-size of a computation is the ratio of the amount of work and the amount of datacommunication involved in the computation. For a particular architecture these amounts can be translated into durations. The computation- and communication performance of an architecture together with the grain-size of computations determine if parallel evaluation may reduce the overall execution time compared to sequential evaluation.

f) locality

An issue closely related to grain-size is *locality*. Two kinds of locality may be distinguished: locality in time and locality in space. In relation to a single computation these concepts may be informally defined in the following way:

When a computation remains active for a relatively long period of time, its locality in time is said to be relatively high. Thus, locality in time corresponds to the amount of work involved in a computation. On the other hand, when a computation only uses data located in a relatively small area of the storage space, its locality in space is said to be relatively high. Thus, locality in space is inversely proportional to the amount of datacommunication in a parallel architecture. The two types of locality in relation to a single computation will not be included in our reference framework because they are already captured by the grain-size property.

Locality in space can also be defined in relation to a group of parallel activities. A group of computations has a high locality in space when most of their activities are confined to a relatively small area of storage space. The latter type of locality will be included as a separate property in the framework of our comparison for the following reason:

When a group of fine grain computations with a high locality in space is executed in the local memory of a single processor they may together form an activity of coarse grain size. If an architecture succeeds to establish this kind of locality, even fine grain calculations may be executed efficiently on a distributed memory architecture.

The most efficient implementations of reduction use a graph representation of the program. An interesting question is wether a collection of rewrite actions on the graph posses some form of locality in space. The Flagship project is based on the assumption that such a locality in space can be established by the run-time system.

In our approach to parallel reduction locality in space is simply enforced by copying an annotated coarse-grain expression to a contiguous remote storage area. The cost of duplicating the expression has to be weighted against the benefits of parallel evaluation.

g) Loadbalancing

Loadbalancing is a mechanism in parallel architectures to obtain an even distribution of parallel activities (load) over the available processing power. Loadbalancing information may be calculated at compile-time, possibly with the aid of programmer annotations. In contrast, distribution of parallel tasks may be computed at run-time. The run-time scheduling of tasks can be decided in one logical centre or, alternatively, the computation of scheduling may be implemented in a distributed fashion. The latter type of loadbalancing is often used in the form of *diffusion scheduling*, where only local criteria are used to migrate tasks.

Summarizing the following aspects may be distinguished: loadbalancing annotations, compile-time loadbalancing, centralized run-time loadbalancing, distributed run-time loadbalancing.

3.4 Comparison of parallel reduction machines

In the next subsections a number of parallel reduction machines will be discussed and a tabular overview of this discussion is presented in figure 10. For each property of the reference framework (*a*-*g*) a separate table is shown. In addition table *h* summarizes the properties of the lower levels of the framework, referring to the figures presented in section 3.2. Table *i* is included to give an impression of the present state of performance analysis in the various projects.

3.4.1 Mago's FP-machine

A proposal for an architecture based on string reduction is Mago's FP-machine [MAG79, MIL89]. Reduction is performed in applicative order. Therefore the disadvantage of string reduction is alleviated somewhat, because no duplication of work will occur when strings are copied. Still large data structures may be copied, where sharing could have been used. An advantage is that some of the copying may be performed in parallel.

The reduction mechanism is based on a fixed set of combinators, proposed as the FP-language by Backus [BAC78]. The language FP, as a consequence of being a combinator language, uses no variables. An FP program can be viewed as one combinator expression applied to a data-structure.

In contrast to the elementary combinators used by Turner, the FP-combinators are quite complex. They are supposed to be sufficient as a programming language. From an architectural point of view the FP-combinators are well suited for special parallel hardware implementations. The rewriting actions look like vector instructions. It was part of the philosophy of the design of FP to specify large computations as basic machine instructions. The Mago-machine may work well when large data-structures have to be processed.

The PMS architecture of the FP-machine is a collection of reduction processors (called leaf processors) interconnected by a tree-shape network. The leaf processors contain the expression string that has to be reduced. The internal nodes of the tree are also active processors supporting simple state transition functions. For instance, the detection of reducible expressions takes place in the network. The top-node of a sub-tree that spans a reducible expression knows that it is responsible for the rewriting of the expression. This node will control all actions that are necessary for the rewriting. It uses the sub-tree to gather and broadcast arguments to the right places. The broadcasting mechanism in the tree of processors allows an expression to be copied to many places in one action.

Computations in Mago's FP-machine will exhibit much locality. The string reduction mechanism creates locality by copying expressions to locally confined regions (sequences of leaf processors). In addition the garbage collection scheme increases locality by squeezing garbage out of strings of leaf processors. This is accomplished by shifting the contents of processors into adjacent unused (or garbage) leaf processors.

Locality increases the performance of the Mago-machine. Reduction will speed up if a reducible string is confined to a short sequence of leaf processors. This is because the spanning tree of the string will be relatively small and few communication hops will be needed.

3.4.2 The AMPS machine

The AMPS architecture (Applicative Multi Processor Architecture) is one of the first machines that performs coarse-grain normal order graph reduction [KEL79]. Parallelism is triggered by strict functions. However, only needed arguments with a sufficient coarse grain size will be evaluated in parallel. The application of a user-defined function is considered to be the right grain of parallelism. The disadvantage of this approach is that the grain-size of user-defined functions may vary considerably.

Reduction is based on graph rewriting. Only a fixed set of primitive rewrite-rules (Flow Graph Lisp, FGL) is supported. User-defined rewrite rules may be composed from the FGL rules.

One of the FGL-rules is the "invoke"-function, which rewrites itself into a pre-defined FGL-graph, representing a user-defined rule (the one that is invoked). The implementation copies the pre-defined graph to the place where it has been "invoked". In order to perform this copying efficiently, user-defined rules are packed into a contiguous block of nodes. The invoke function is a coarse-grain operation that avoids many memory allocation steps and pointer copying that would occur in a fine grain combinator machine (e.g NORMA, SKIM). One should realize that this idea was published at the same time that Turner published the implementation of SASL based on fine-grain SKI-combinators. It would still take several years before super-combinators were invented.

The AMPS architecture consists of a collection of FGL-processors connected to a tree-shaped network. The network serves two purposes: data-transport and loadbalancing. If the distance between parent and child processes remains small (locality is maintained), the tree topology will transport data efficiently. If locality is lost, messages have to travel high up the tree, and a communication bottle-neck will arise. The loadbalancing mechanism has to maintain the locality. It has not been shown that this is actually the case.

The nodes in the tree network are simple processors. Apart from transporting data, they also monitor the load in their sub-trees. After adding up the load figures reported by its child-trees, a node presents the result to its parent node. All nodes in the tree continuously perform this load calculation. If a node detects a difference in the load of its child trees that exceeds a certain threshold, it initiates the transfer of reduction tasks from the most loaded sub-tree to the less loaded sub-trees.

The AMPS architecture supports one global address space though each FGL-processor only has local memory. The coarse grain-size of tasks should compensate the communication overhead caused by the implementation of a single global address space.

The AMPS architecture contains a number of well chosen architectural features: Only coarse grain reduction tasks are evaluated in parallel, diffusion scheduling is used for loadbalancing, reduction is based on coarse grain graph-rewrite rules and finally sequential tasks are reduced in normal order.

3.4.3 The Rediflow machine

The Rediflow architecture [KEL84, KEL86] has been presented as a follow-up of the AMPS machine. The main improvement is the merging of reduction and data-flow, which explains the name of the architecture. The ideas behind Rediflow can be viewed as an optimization in the implementation of reduction by exploiting the behaviour of streams. Programs based on communicating sequential processes do not run well on architectures exploiting coarse grain strict argument parallelism, like the AMPS (see chapter 7). In a functional language processes are modeled as tail-recursive functions, consuming and producing streams. If this kind of behaviour is recognized by the implementation of the reduction model, efficient imperative code can be generated for the processes. The Rediflow architecture was the first reduction machine proposal in which (stream-based) functions are compiled to imperative code. The issue of implementing communicating sequential processes on our architecture is treated in chapter 7.

An other difference with the AMPS architecture is that it is recognized that a physical tree topology is not desirable for mapping the process tree generated by strict argument parallelism. The Rediflow architecture proposes an XPUTER as the only processing element, incorporating a FGL-processor, a memory and a packet switch. Several topologies may be realized with such an element (N-cube shuffle exchange, grid). The ideas are similar to the architecture of the Transputer [WHI85].

The loadbalancing algorithm has been adapted to a non-tree topology. Still a diffusion scheduling algorithm is used, but loadbalancing information is exchanged between all neighbouring processors. The algorithm computes for each processor a "pressure", derived from the internal load and the pressures of surrounding processors.

Task migration is initiated when a significant pressure difference is detected. The production of parallel tasks is modulated by alternating between a FIFO- and a LIFO task evaluation. This idea was first proposed by Sleep in the ZAPP architecture.

3.4.4 The ZAPP architecture

The ZAPP (Zero Assignment Parallel Processor) architecture [BUR81, MCB87] is similar to Rediflow with respect to the processing elements and the loadbalancing strategy. The communication switch between processing elements is only required to have a "strong connectivity". In contrast to what is suggested in an earlier paper [BUR81], it is clearly stated in [MCB87] that no global address space is supported.

Parallelism is generated by the use of *paradigms*, which may be considered as patterns in algorithms. An example is the divide-and-conquer paradigm, which is the only one currently used in ZAPP. This method is similar to the the approach to parallelism in our architecture. The similarity extends to the way in which a program is distributed over the processing elements and the way in which expressions are copied. The program is initially broadcast to all processing elements, and function applications contained in the divide-and-conquer paradigm are copied to remote processors for parallel evaluation.

The loadbalancing scheme of ZAPP uses the "single-stealing" rule. This rule allows a relatively empty processor to steal reduction tasks from neighbouring processors. However, a stolen task can not be migrated again. This guarantees that the distance between parent and child tasks is at most one communication link. In Rediflow tasks can migrate over an unspecified distance.

As in Rediflow the evaluation order of tasks in a processing element switches between LIFO (breadth first) to FIFO (depth first), depending on the load of the processor. The number of tasks grows linearly with the depth of the process tree during depth-first evaluation, whereas it grows exponentially during breadth-first evaluation.

No reduction model is yet specified in the ZAPP proposal. Consequently no decisions have been made with respect to program representation, reduction mechanism and garbage collection. The application programs used in the reported experiment [MCB87] are entirely programmed in OCCAM (in a functional style).

3.4.5 The Alice and Flagship machines

Fine grain parallel graph reduction is proposed in the Alice architecture [DAR81, EIS87]. The program is represented as a graph consisting of nodes that are called *packets* to indicate the transportable nature of the nodes. A packet in the Alice machine is similar to a node in the graph rewrite language CLEAN [BRU87]. It contains a function name and pointers to the arguments of the function. If a rewrite rule exists for the function and the arguments have the form required by this rule, the packet can be rewritten as specified by the rule. Such a packet is called a rewritable packet. The granularity of parallel actions in the Alice architecture is a single rewrite action. Rewritable packets are stored in a special pool, from which a fixed number of parallel tasks retrieve packets to be rewritten.

The Flagship design [WAT86, WAT87] is similar to the Alice proposal. Much attention is paid to the grain-size and communication cost of packet rewrite actions. Packet rewrite rules are program-derived combinators, translated to efficient imperative code. Before a processor attempts rewriting a packet, separate concurrent processes take care that all direct descendants of the packet to be rewritten are copied to the local memory of the processor. If this copying can be done faster than the rewriting, all processors can effectively reduce at their highest speed.

The reduction order of both Alice and Flagship is in principle applicative, but with some difficulty also normal order can be supported: Special packet types are introduced to support unevaluated and curried applications. The Flagship machine will make use of strictness information inside rewrite rules to construct data-flow graphs and save the overhead to detect unevaluated packets on strict argument positions.

Both Alice and Flagship architectures rely on a task diffusion mechanism that dynamicly moves reducible sub-graphs between processors in order to keep the load balanced. If such a mechanism can be efficiently implemented on a local memory architecture using rather fine-grained rewrite actions, is a question that remains to be answered.

The Flagship proposal mentions weighted reference-counting as the garbage collection mechanism that will probably be used [WAT86]. Cycles will be recovered by an additional mark-and-scan sweep from time to time.

3.4.6 The GRIP machine

As opposed to all previous architectures, the GRIP (Graph Reduction In Parallel [PEY87b, CLA86]) machine supports a global address space in hardware by using a single shared connection between the processors in the system. The practical implementation has already been discussed in 3.2.4. The GRIP machine is intended to exploit medium to coarse grain parallelism. The architecture is claimed to hold a position in between a tightly- and loosely coupled system. In the design this is reflected by the packet-switched operation of the shared bus connection. Operations on this bus have a coarser granularity than single word at the time accesses. On the other hand the granularity is smaller than complete graph rewrite actions.

The reduction mechanism is based on compiled graph-rewrite rules. A mark-and-scan garbage collection mechanism is used. Reduction is stopped in each processor to execute the marking phase. Scanning is performed concurrently with reduction.

Strict functions give rise to new reduction tasks. Information about the strictness of functions and applications is supposed to be present in the graph. Strictness analysis and user annotation are mentioned as possibilites to derive this information [CLA86]. Reduction processes "spark" new tasks when strictness tags are encountered during unwinding or rewinding the spine of function applications.

It remains unclear if the grain size of these tasks is controlled. Because only strictness is mentioned as a criterion to spark tasks it can be assumed that task granularity will be fine. It is suggested that the synchronisation of tasks is implemented as basic operations of the specialised memory processors (see section 3.2.3).

The amount of parallelism is managed by changing the evaluation order in the distributed task pool between FIFO and LIFO like in the previous architectures. An additional "resumed-first" strategy is mentioned which gives priority to recently unblocked tasks over new tasks that have not been active yet.

3.4.7 The APERM machine

The parallel reduction model that we use in our architecture (the Amsterdam Parallel Experimental Reduction Machine) is based on a combination of graph- and string reduction. Within each of the local storage spaces pure graph reduction is performed. However, when subgraphs are detected that represent coarse grain computations, they may be copied to a remote storage space to benefit from parallel evaluation. This copying is a kind of string reduction, but a special reduction strategy avoids duplication of work in these cases. This reduction mechanism is a unique feature of APERM.

Locality in space is enforced by copying subgraphs to a separate remote storage space. Reduction of the copied subgraph can be completed without reference to the original graph. In particular, garbage can be collected locally without interference with the other reduction processes.

In APERM the transport of subgraphs occurs without synchronization to the reduction processes and after reservation of sufficient contiguous storage space at the destination. This yields definite advantages with respect to communication speed, as discussed in chapter 8. To implement this kind of graph transport a copying garbage collection scheme is mandatory. None of the other reduction machine proposals consider unsynchronized communication support for graphs. Flagship, Alice and GRIP base their data-communication on much finer grain units than a subgraph. The ZAPP architecture has not yet implemented graph reduction and consequently specialized graph-communication has not been considered.

Parallelism in APERM is generated by annotated strict functions. The annotations are inserted into the source text by the programmer. The granularity and the amount of parallel computations is controlled by a threshold mechanism at the source text level of the application program. The mechanism is incorporated in the application by means of program transformations (see chapter 5).

The only reduction machine based on a similar approach (i.e. annotated parallelism) is ZAPP. However, the grain size of computations is not restricted to a minimum and the amount of parallelism is supposed to be managed by the run-time system (the FIFO/FIFO strategy).

In APERM loadbalancing is performed by a centralized process. Based on run-time acquired knowledge of the structure of application programs (the execution profile), this loadbalancing algorithm may obtain better results than the diffusion schemes employed in all other reduction architectures reviewed in this section. In addition the loadbalancing algorithm achieves a considerable optimization of the communication cost by exploiting the partly overlapped address spaces of APERM.

framework properties	Program representation	Reduction mechanism	Garbage collection
MAGO	nested delimited strings	string reduction of fixed set of combinators	asynchronous shift operations between the leave-processors
AMPS	graph	graph rewriting of a fixed set of primitive functions	not specified
Rediflow	not specified	graph rewriting of a fixed set of primitive functions, stream-based user-defined functions compiled to imperative machine code.	not specified
ZAPP	graph	not specified	not specified
Flagship	graph	graph rewriting of user-derived combinators, compiled to imperative machine code.	reference counting. Cycles recovered by additional mark-and-scan
GRIP	graph	graph rewriting of user-derived combinators, compiled to imperative machine code.	mark-and-scan. Scanning is performed concurrently with reduction
APERM	locally graph & globally tree	locally graph rewriting, globally a kind of string reduction. Graph rewriting of user-derived combinators, compiled to imperative machine code	copying garbage collection. No synchronization of reduction processes required

framework properties	Abstract machine model	PMS architecture	Performance evaluation method	Application programs used
MAGO	Collection of active strings	Tree of processors	simulation	??
AMPS	figure 2	Tree of processors	not specified	not specified
Rediflow	figure 2	figure 3b	simulation	not specified
ZAPP	figure 5	figure 3b	measurements on transputer hardware	small to medium sized programs
Flagship	figure 2	figure 3a	prototype being constructed	not specified
GRIP	figure 6	figure 7	prototype being constructed	not specified
APERM	figure 8	figure 9	hybrid simulation on prototype	small to medium sized programs

frame-work properties	Reduction Strategy:	Grain size	Locality	Loadbalancing
MAGO	applicative order, innermost parallel	fine grain machine instructions	not exploited	no load-balancing
AMPS	normal order reduction, strict argument parallelism	application user-defined function. (fine-coarse grain)	supposed to be maintained by task diffusion algorithm	task diffusion, by a pressure algorithm
Rediflow	normal order reduction, strict argument parallelism	application user-defined function (fine-coarse grain)	supposed to be maintained by task diffusion algorithm	task diffusion, by a pressure algorithm. The amount of parallelism is controlled by alternating between a LIFO- and FIFO task evaluation order
ZAPP	sequential strategy not specified, strict argument parallelism of annotated functions	indicated by the programmer (coarse grain)	maintained by the "single-stealing" rule	task diffusion, by "single-stealing" rule. The amount of parallelism is controlled by alternating between a LIFO- and FIFO task evaluation order.
Flagship	applicative order, but normal order is also possible	packet rewrite operations. (fine grain)	maintained by task diffusion and caching remote parts of the graph	task diffusion mechanism
GRIP	normal order, strict argument parallelism	application of user-derived combinator (fine-coarse grain)	not specified, perhaps not exploited	not specified. The amount of parallelism is controlled by alternating between a LIFO- and FIFO task evaluation order. Also "resumed first" strategy
APERM	normal order, strict argument parallelism of annotated functions	annotated coarse grain expressions	enforced by copying annotated coarse grain expressions	central process controls loadba-lancing with heuristics based on execution profile. The amount of parallelism is controlled by a threshold mechanism

Figure 10: A comparison of several reduction machines referring to the framework of section 3.3.

3.5 Dual ported modules in a bus oriented architecture

The prototype of APERM has been constructed from commercial available processor- and memory boards. Each processor board is equipped with a reasonable amount of local memory, whereas off-board memory can be accessed via a bus interface. The particular boards that we use are equipped with two separate bus interfaces (VME and VMX), which implies that we have dual ported processors (DPP) and dual ported memories (DPM). For processor boards the available off-board address space is divided in two parts where each part will access a different bus. Memory boards can be accessed via both ports simultaneously. To assure mutual exclusion of simultaneous operations a DPM contains an asynchronous arbiter.

The architecture of APERM uses DPM's to implement fast communication links between adjacent processors. In chapter 4, two advantages of such an architecture are pointed out. The first practical advantage is that two-points arbitration in DPM's is faster than multi-point arbitration on a bus. The second advantage is that DPM's provide a shared memory between two processors, which offers opportunities to minimize communication cost.

In this section we show in addition to these advantages that a bus oriented architecture based on DPM's and DPP's can achieve a double performance compared to a similar architecture using single ported modules.

In practical bus oriented architectures each processor module has a restricted amount of fast local memory (on-board) at its disposal. For fundamental reasons, the access time to this local memory is (much) shorter than the access time to the large off-board memory. When software is implemented on such an architecture the highest speed will be obtained when those parts of data and code that are most frequently accessed, are loaded into the local memory. For an implementation of graph reduction optimal results are obtained when only the graph is stored off-board, whereas the code of the reducer and the various stacks are kept in the on-board memory.

Under these conditions it can be shown that a large fraction of all memory cycles will refer to the local memory [HAR89]. On the one hand, all instruction fetches and stack references will access the local memory. On the other hand, the optimization techniques used in the compilation of reduction, will use local stack-evaluation in favour of graph reduction whenever possible.

As a consequence of this situation, the off-board memory bandwidth is under-utilized when only a single processor is performing graph reduction. More reduction processors can be added to the same bus, until either the bus or the memory is saturated. This situation is illustrated in figure 11a. The off-board memory space is constituted by three storage modules, while three reduction processors are connected to these memories by a single bus.

To demonstrate the performance gain of a factor of two by exploiting DPM's and DPP's we make three simplifying assumptions (A1-A3). In the example it is assumed that the three reduction processors saturate the off-board memory bandwidth (A1). Bus speed and bus-arbitration overhead are neglected. Thus off-board memory bandwidth is considered to be the only limiting factor (A2).

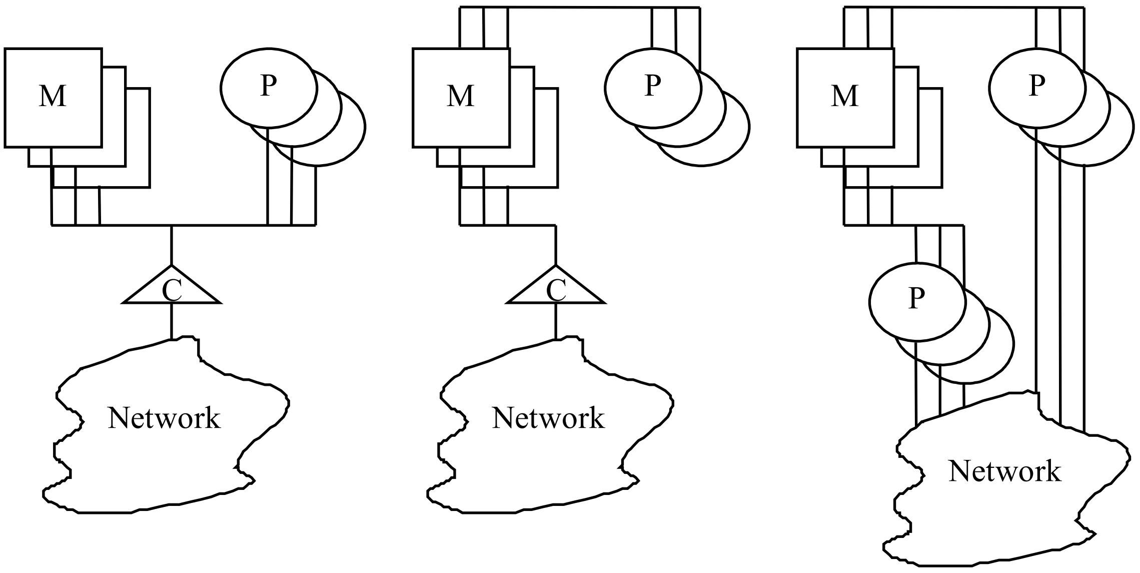


figure 11a figure 11b figure 11c

Figure 11: Configurations with dual ported memories.

In figure 11a, the cluster of reduction processors is connected to other clusters by a network. The precise implementation of this network is left unspecified except for the the presence of a communication processor. We assume that this processor continuously transports subgraphs between clusters at a speed that saturates the off-board memory bandwidth (A3). In our job-based reduction model (chapter 5) such a processor is feasible (see chapter 8). Now it becomes clear that the performance of graph reduction drops by a factor of two, when both the reduction processors and the communication processors saturate the memory bandwidth.

Figure 11b presents a configuration where both reduction and communication can operate simultaneously at full speed. The memory modules are provided with a second port and a separate bus connects them to the reduction processors. The dual ported memories of figure 11b have the same bandwidth as the single port memories of figure 11a (the two ports are served by time-multiplexing the accesses from both sides). Consequently one memory module can only serve one bus at full speed. However, when reduction processors and communication processor access different modules, they may operate simultaneously at full speed. To make sure that such a situation frequently arises, the storage allocation algorithm can take care of the even distribution of graph nodes across the available memory modules. In the architecture of figure 11b the combination of interleaved memory modules and dual ported time-multiplexed access provide a twofold performance improvement over the conventional architecture of figure 11a.

Still another improvement can be achieved if the communication processor of figure 11b is replaced by reduction processors as shown in figure 11c. These processors can be programmed to perform both reduction and sub-graph communication. When the communication need in the whole system is not capable to saturate the off-board memory bandwidth, some of the communication processors may be switched to perform reduction tasks. This is not possible in figure 11b, and consequently a twofold performance increase with respect to figure 11a can now be maintained even when communication traffic drops to zero. Under these circumstances the architectures of figure 11a and 11b would have the same performance.

From the symmetry of the architecture of figure 11c, one can observe that all processors may be used for either reduction or communication. Moreover, if we impose certain restrictions on the communication network, each processor is capable to perform reduction in two clusters. This is because in figure 11c all processors are equipped with two ports, one into the own cluster and one connected to the network. When the latter connections are directly tied to DPM's in other clusters, each processor can perform reduction in two address spaces corresponding to both its ports. These restrictions turn the network into a store-and-forward network, which would be no disadvantage in the context of extensible architectures. The future architecture of APERM as discussed in chapter 4, corresponds to the structure of figure 11c. In the same chapter the advantages of this structure concerning the optimization of communication are outlined.

References

[BAC78] J. Backus, "Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs", CACM, Vol 21, No 8, Aug 1978, pp 613-641.

[BAR84] H.P. Barendregt, "The lambda calculus, its syntax and semantics", North Holland, Amsterdam, 1984

[BAR87] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, M.R. Sleep, "Term graph rewriting", in Conf. on Parallel Architectures and Languages Europe (PARLE), part II, Eindhoven, the Netherlands, LNCS vol 259, pp. 141-158, June 1987.

[BEE89] M. Beemster, L.O. Hertzberger, H.L. Muller, E. Brandsma, B.J.A. Hulshof, A.C.M. Oerlemans, "Implementation aspects of the PRISMA parallel main memory database machine", to appear in IWDM, Nancy, 1989.

[BEL71] C.G. Bell, A. Newell, "Computer structures: readings and examples", McGraw Hill, 1971.

[BEL79] C.G. Bell, J.C. Mudge, J.E. McNamara, "Computer engineering", fourth press, Digital Press, 1979

[BER75] K.J. Berkling, "Reduction languages for reduction machines", in Second Ann. Symp. on Computer Architecture, Jan 1975, pp 133-140

[BRUI72] N.G. de Bruijn, "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation", Indag. Math. 34, 1972, pp 381-392.

[BRU87] T.H. Brus, M.C.J.D. van Eekelen, M.O. van Leer, M.J. Plasmeijer, "Clean: a language for functional graph rewriting", in Third conf. on functional programming languages and computer architecture, Portland , Oregon, USA, LNCS 274, pp. 364-384, September 1987.

[BUR81] F.W. Burton, M.R. Sleep, "Executing functional programs on a virtual tree of processors", in Conf. on functional languages and computer architecture, ACM, Porthmouth, New Hampshire, 1981, pp 187-194.

[CLA86] C. Clack, S.L. Peyton Jones, "The four stroke reduction engine", in ACM Conf. on Lisp and functional programming, Boston, Aug 1986, pp 220-232

[CLAR80] T.J.W. Clarke, P.J.S. Gladstone, C.D. MacLean, A.C. Norman, "SKIM - the S,K,I Reduction Machine", Proceedings of the 1980 ACM Lisp Conference, pp. 128-135, August 1980.

[COU85] G. Cousineau, P. Curien, M. Mauny, "The categorial abstract machine", in Second conf. on functional programming languages and computer architecture, Nancy, Sept. 1985, LCNS 201, pp 50-64.

[DAR81] J. Darlington, M. Reeve, "ALICE: A multiple-processor reduction machine for the parallel evaluation of applicative languages", in Conf. on functional programming, languages and computer architecture, pp 65-76, ACM, Porthmouth, New Hampshire, Oct. 1981.

[DAV84] R. Davis, D. Thomas, "Systolic array chip matches the pace of high speed computing", Electronic Design, Oct 1984, pp 207-218

[EEK88] M.C.J.D. van Eekelen, "Parallel graph rewriting - some contributions to its theory, its implementation and its application", Ph. D. thesis, Dept. of Comp. Sci., Univ. of Nijmegen, Dec 1988.

[EIS87] S. Eisenbach (editor), "Functional Programming: languages, tools and architectures", Ellis Horwood Limited, Chichester, 1987

[FAI87] J. Fairbairn, S. Wray, "Tim: a simple, lazy abstract machine to execute supercombinators", in Third conf. on functional programming languages and computer architecture, Portland , Oregon, USA, LNCS 274, pp. 34-45, September 1987.

[GUR87] J. Gurd, W. Böhm, Y.M. Teo, "Performance issues in dataflow machines", Future Generations Computer systems, Vol 3, No 4, Dec 1987, pp 285-297.

[HAR89] P.H. Hartel, "Performance analysis of storage management in combinator graph reduction", Ph. D. thesis, Dept. of Comp. Sys., University of Amsterdam, 1988

[HUD85] P. Hudak, B. Goldberg, "Distributed execution of functional programs using serial combinators", IEEE Transactions on computers, Vol. C-34, No. 10, pp 881-891, Oct. 1985.

[HUG82] R.J.M. Hughes, "Super combinators - a new implementation method for applicative languages", in ACM symp. on Lisp and functional programming, Pittsburg, Aug. 1982, pp 1-10

[JOH84] T. Johnsson, "Efficient compilation of lazy evaluation", Proc. of the ACM Sigplan '84, Sigplan Notices, Vol. 19, No 6, June 1984, pp 58-69.

[JOH85] T. Johnsson, "Lambda lifting: transforming programs to recursive equations", in Second conf. on functional programming languages and computer architecture, LNCS 201, Nancy, Sep. 1985, pp 190-203

[KEL79] R.M. Keller, G. Lindstrom, S. Patil, "A loosely-coupled applicative multi-processor system", AFIPS National computer conf. proc., Vol 48, New York, June 1979, pp 613-622

[KEL84] R.M.Keller, F.C.H.Lin, "Simulated performance of a reduction based multi-processor", IEEE computer, Vol. 17, No. 7, pp 70-82, July 1984

[KEL86] R.M. Keller, J.W. Slater, K.V. Likes, "Overview of Rediflow II Development", Proceedings of a workshop on graph reduction, September/October 1986, Santa Fé, New Mexico, USA, LNCS 279, pp 203-214

[KIE85] R.B. Kieburtz, "The G-machine: A fast, graph-reduction evaluator", in Second conf. on functional languages and computer architecture, Nancy, LCNS 201, pp. 1 - 16, September 1985

[KLU83] W.E. Kluge, "Cooperating reduction machines", IEEE Transactions on computers, Vol. C-32, No. 11, pp 1002-1012, Nov. 1983.

[MAG79] G.A. Magó, "A network of microprocessors to execute reduction languages", Part I and II, International journal of computer and information sciences, Vol. 8, No. 5, pp 349-385 Oct. 1979 and No. 6, pp 435-471. Dec. 1979.

[MCB87] D.L. McBurney, M.R. Sleep, "Transputer-based experiments with the ZAPP architecture", in Conf. on parallel architectures and languages Europe (PARLE), part I, Eindhoven, the Netherlands, LNCS 259, pp. 242-259, June 1987.

[MIL89] V.M. Milutinovic (editor), "High-level language computer architecture", Computer Science Press, New York, 1989

[NGU88] T.M. Nguyen, V.P. Srini, A.M. Despain, "A two-tier memory architecture for high-performance multiprocessor systems", Proc. of the 1988 ACM int. conf. on supercomputing, Saint-Malo, France, Juli 1988.

[ODIJ85] E.A.M. Odijk, "DOOM: a Decentralized Object-Oriented Machine", Doc. Nr. 0125, Esprit 415 internal report, Philips, Eindhoven, 1985.

[ODIJ87] E.A.M. Odijk, "The DOOM system and its applications: A survey of Esprit 415 subproject A, Philips research laboratories", in PARLE Parallel architectures and languages Europe, Eindhoven, 1987, Part I, LNCS 258, pp 461-479.

[PEY87a] S.L. Peyton-Jones, "The implementation of functional programming languages", Prentice Hall, Englewood Cliffs, New Jersey, 1987

[PEY87b] S.L. Peyton-Jones, C. Clack, J. Salkild, M. Hardie, "GRIP-a high performance architecture for parallel graph reduction", in Third conf. on functional programming languages and computer architecture, Portland , Oregon, USA, LNCS 274, pp. 98-112, September 1987.

[PFI85] G.F. Pfister, W.C. Brantley, D.A. George, S.L. Harvey, W.J. Kleinfelder, K.P. McAuliffe, E.A. Melton, V.A. Norton and J. Weiss, "The IBM research parallel processor prototype (RP3): introduction and architecture", in proc. of 1985 Int. conf. on parallel processing, IEEE Computer society press, 1985.

[ROB65] J.A. Robinson, "A machine-oriented logic based on the resolution principle", Journ. ACM, Vol 12, No 1, Jan 1965, pp 23-41

[RUG87] C.A. Ruggiero, J. Sargeant, "Control of parallelism in the Manchester dataflow machine", in Third conf. on functional programming languages and computer architecture, Portland , Oregon, USA, LNCS 274, pp. 1-15, September 1987.

[SCH86] M. Scheevel, "NORMA: A graph reduction processor", ACM conference on LISP and functional programming, M.I.T., Aug 1986

[SCHÖ24] M. Schönfinkel, "Über die Bausteine der mathematichen Logik", Mathematische Annalen, Vol. 92, No. 6, 1924, pp 305-316

[SHU85] M.J. Shute, P.E. Osmon, C.L. Hankin, "COBWEB, A combinator reduction Architecture", in Second conference on functional programming languages and computer architecture, Nancy, LCNS 201, Sept. 1985, LCNS 201, pp 99-112.

[TAN84] A.S. Tanenbaum, "Structured computer organisation", second edition, Prentice Hall, Englewood Cliffs, New Jersey, 1984.

[TUR79] D.A. Turner, "A new implementation technique for applicative languages", Software practice and experience, Vol. 9, No. 1, pp 31-49, Jan. 1979

[TUR85] D.A. Turner, "Miranda: A non-strict functional language with polymorphic types", in Second conf. on functional languages and computer architecture, Nancy, LCNS 201, pp. 1 - 16, September 1985.

[VRE88] W.G. Vree, P.H. Hartel, "Parallel graph reduction for divide and conquer applications - part I", PRM project internal report D-15, Dept. of Comp. Sys., Univ. of Amsterdam, December 1988.

[WAD71] C.P. Wadsworth, "Semantics and Pragmatics of the Lambda-calculus", Ph. D. thesis, Oxford University, U.K., 1971.

[WAT86] I. Watson, P. Watson, "Graph reduction in a parallel virtual memory environment", Proceedings of a workshop on graph reduction, September/October 1986, Santa Fé, New Mexico, USA, LNCS 279, pp 265-274

[WAT87] P. Watson, I. Watson, "Evaluating functional program on the FLAGSHIP machine", in Third conference on functional programming languages and computer architecture, Portland , Oregon, USA, LNCS 274, pp. 80-97, September 1987.

[WHI85] C. Whitby-Strevens, "The Transputer", in 12th Ann. int. symp. on computer architecture, June 1985, pp 292-300

[WIL88] R. Wilson, "Motorola unveils new risc microprocessor flagship", Computer Design, May 1988, pp 21-32

[XER81] The Xerox learning research group, "The Smalltalk-80 system", Byte, Vol 6, No 8, Aug 1981, pp 36-48

CHAPTER IV

A COARSE GRAIN PARALLEL ARCHITECTURE FOR FUNCTIONAL LANGUAGES[1]

[1] in Conf. on Parallel Architectures and Languages Europe (PARLE), Eindhoven, the Netherlands, LNCS vol. 365, pp 269-285, June 1989

A COARSE GRAIN PARALLEL ARCHITECTURE FOR FUNCTIONAL LANGUAGES

L.O. Hertzberger, W.G. Vree
Department of computer systems, University of Amsterdam
PO box 41882, NL-1009DB Amsterdam

abstract

Design considerations of a coarse grain parallel architecture for functional languages are presented. These include extensibility, the separation of computation and control of parallelism, the introduction of partially shared memories, a cluster concept and a conceptually centralised loadbalancing mechanism. The implementation of parallel reduction is based on annotation of coarse grain strict arguments. Speed-up figures for a number of application programs are obtained by measurements on a pilot implementation of the architecture. The experience obtained with the experimental machine suggests the use of VLSI for specialised parts of the implementation. The proposed design is compared with related architectures.

1. Introduction

Within the context of the Dutch Parallel Reduction Machine Project [BAR87] an experimental machine architecture has been developed, to gain experience with a number of architectural concepts, which are well suited for the implementation of functional languages. Because of the lack of side effects in these languages, it is possible to annotate independent coarse grain subexpressions, and subsequently distribute these expressions as parallel tasks over a machine. We discuss the implementation of parallel reduction on our architecture, which is referred to as APERM, the Amsterdam Parallel Experimental Reduction Machine.

Functional languages are interesting candidates for programming parallel machine architectures. For example, functional programs can be converted by the technique of lambda lifting [JOH85] into a collection of (super) combinators. These (super) combinators form a sound basis for parallel computation [HUD85] because they can in principle be evaluated as separate units of local computation, without the need to access a global environment.

An important aim of the research in parallel computing is the development of extensible multiprocessor architectures. These architectures have a regular structure such that the number of processing elements can be increased without the need to change the already existing part of the machine. Only extensible machines will be able to keep pace with the progress in semiconductor integration technology. For this reason it was decided that the APERM should have a regular structure.

One of the consequences of this decision was that locality became a vital issue in the design of both hard -and software components of the machine. In hardware locality constraints the design of the storage -and communication system. In software the granularity of processes dominates the development of the reduction model and the application programs. Both aspects of locality have been the focus of our research effort that resulted in APERM. To support this research it was decided to build a prototype reduction machine with commercial available processing components, that could provide us with sufficient measurement data concerning locality. In a second phase of the project we intend to optimise the machine by incorporating special components for reduction-specific tasks.

2. Separation of reduction and parallelism

A program written in a functional language is first translated into a set of rewrite rules (e.g. supercombinators) and a main expression. The main expression is then repeatedly rewritten, according to the given rules, until no more rewriting is possible. This result is called *normal form* and the process of rewriting is called *reduction*.

There are two different approaches to implement reduction on a parallel architecture. The first possibility is to have one global parallel reducer. This reducer keeps track of all rewritable subexpressions (R's in figure 1), and schedules the rewriting of these expressions on the available processors. Consequently this reducer has to know about both parallelism and reduction. The second possibility (see figure 2) is to have an orchestra of sequential reducers directed by a separate conductor, where the reducers have no knowledge about parallelism and the conductor has no knowledge of reduction. Communication between conductor and reducer is restricted to the exchange of reducible expressions and their normal forms. We considered the second possibility more promising to realise an extensible architecture by the exploitation of locality. It has the advantage that the orchestra of sequential reducers forms a collection of local computations of coarse granularity. On the contrary, the first possibility is based on parallel scheduling of single rewrite actions, which might turn out to have too fine a granularity.

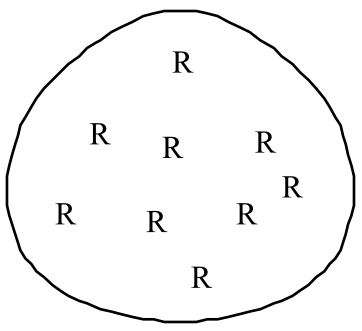


Figure 1: parallel reducer

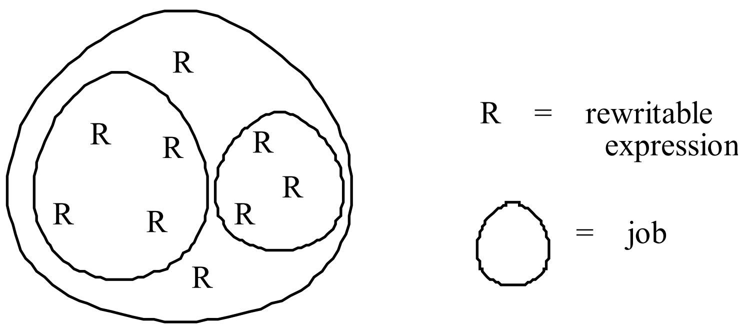


Figure 2: orchestra of sequential reducers

The choice to separate the control of parallelism and reduction also appeared to be fruitful in another area. During the progress of the project, major improvements have been achieved in the way that sequential reduction is implemented. Compilation techniques have been established for functional languages [JOH84, FAI87], which produce very efficient sequential machine code and allow functional programs to compete with and even surpass imperative languages. Our machine can benefit from these achievements in sequential reduction, because the research in reduction and parallelism is separated.

2.1 Abstract machine model

Separation of reduction and parallelism not only gives the possibility to study both problems independently it also leads in a natural way to a model for our reduction machine. In this model the reduction tasks are performed by a pool of reducer processes, which evaluate coarse-grain entities representing the functional program. How these entities are generated is discussed in section 2.2.

The problem of controlling the granularity of parallelism is solved in our model at the level of the application program. By a program transformation conditional statements are inserted that compare the grain-size of parallel tasks with a fixed threshold parameter. The application programmer has to provide a measure for the grain-size of parallel tasks, e.g. the size of a data-structure. When the grain-size of parallel tasks drops below the threshold they are evaluated sequentially.

The distribution of parallelism is controlled by a conceptually centralised process that we call the conductor. It allocates pieces of work to different reducers and tries to balance the load of processors by monitoring the usage of storage- and process resources in the system.

2.2 Parallel reduction model

Normal order graph reduction is the basis of our reduction model. The graph representation of a functional program, allows common subexpressions that arise during reduction to be shared.

As a consequence shared subexpressions are only evaluated once. This is an important optimisation compared to string reduction, where duplication of reducible expressions also means duplication of work.

The graph representation of functional programs needs a single global address space. Therefore most parallel reduction machines feature a globally addressable storage space [PEY87, WAT87]. When a global address space is implemented on top of an architecture with distributed storage, memory can no longer be accessed in constant time. The access costs depend on the size of the machine and can amount to a considerable overhead. In the implementation of the reduction model on APERM we have chosen only to support locally addressable memory spaces. Therefore we have to provide an explicit way to distribute parts of the graph structure across the local memories.

2.2.1 Job based parallel reduction

The pieces of work controlled by the conductor are coarse grain subexpressions, that we call *jobs*. In the machine these subexpressions are represented by graphs, so at the implementation level a job is a subgraph . No attempt has been made yet in the project to develop heuristic methods to detect such jobs at compile-time, but we will consider this possibility in the future. At present the programmer has to annotate those subexpressions that represent a sufficient amount of computation to be treated as a job by the conductor. These annotations do not change the meaning of the program, but merely cause the reducer to inform the conductor about the presence of a potential job. The conductor, on its turn makes the decision if parallel reduction is possible and if so, takes care that the job is transmitted.

The idea is that jobs are completely copied to a remote processor when their evaluation is needed. The property of graph reduction to share subexpressions can thus not be maintained on the job level. Within jobs pure graph reduction is performed, but when parallel evaluation is performed shared subexpressions are unshared by copying them to the remote processor.

Summarizing, our parallel reduction model employs a kind of string reduction on the global job level and graph reduction within jobs.

2.2.2 Sandwich reduction strategy

To avoid the major disadvantage of copying (duplication of work), a special reduction strategy has been devised on the job level [VRE88]. This strategy guarantees that a job will only contain one reducible expression, starting at the top node of the job graph. We call this reducible expression the primary redex of the job. When the job is transported to another processor, it is guaranteed that there are no other (secondary) redexes . Thus, when copying takes place, no work can be duplicated.

In practice the sandwich strategy has been implemented by a special function to be used by the programmer to annotate parallel jobs:

$$\text{sandwich } G\ job_1\ job_2\ \ldots\ job_n$$
$$\text{where } job_i = F_i\ a_{i1}\ a_{i2}\ \ldots\ a_{im}$$

The sandwich construct is the only means in the language to create jobs. An expression is sequentially reduced in normal order until a sandwich expression is needed. The reduction of G $job_1\ job_2\ ...\ job_n$ is then suspended until the parallel evaluations of $job_1\ job_2\ .\ .\ .\ job_n$ have been completed. However, before the jobs are dispatched for parallel evaluation, all arguments $a_{i1}\ a_{i2}\ .\ .\ .\ a_{im}$ of each job_i are sequentially reduced to normal form, because subexpressions shared between the a_{ij} may contain redexes. After normalisation of the a_{ij} only normal forms are shared. Now copying the normal forms of the a_{ij}, in order to ship the jobs, cannot result in extra work[1]. Similarly we require that the F_i do not contain reducible expressions.

The strategy has been called "sandwich strategy" because it contains one level of applicative evaluation between two levels of normal order evaluation. We have embedded the sandwich function in SASL [HAR88].

Care should be taken to avoid non-termination of the program when sandwich functions are inserted. The programmer has to ensure that all arguments a_{ij} represent terminating computations. The sandwich strategy cannot directly exploit "pipe-line" parallelism, where a chain of processes transform a list of input values while each process only operates on one element of the list at a time. However, it is possible to transform a large class of programs based on pipe-line parallelism into versions that can be annotated with the sandwich function. We have called this transformation *communication lifting* [VRE89]. The tidal model discussed in section 4.1 is an example of a program that was originally expressed as a set of processes interconnected by streams (lists with pipe-line behaviour) and that has been transformed by communication lifting into a version that runs efficiently on our job-based reduction model.

2.3 Control of Parallelism

Jobs generated by sandwich annotations exhibit a strict hierarchical structure. A job executing a sandwich application becomes a parent task that spawns a number of independent child tasks. There will be no communication between the child tasks, because jobs are self contained. The only communication that takes place is the transfer of the child tasks to possibly remote reducers, and the returning of the results to the parent.

The hierarchical task structure avoids problems associated with global garbage collection [HAR88] and gives rise to interesting possibilities for loadbalancing strategies. We have investigated a conceptually centralised algorithm (the conductor) that uses certain heuristics to find a near optimal schedule. The heuristics of this loadbalancing strategy is based on the

[1] provided that for all jobs in a sandwich expression: $job_i \neq job_j$ when $i \neq j$.

assumption that there exists a dependency between the size of the arguments in a job expression and the amount of work represented by the job.

A second task of the conductor is to use the knowledge about the concrete architecture, in our case the presence of overlapping storage spaces (see 3.2), to minimise the communication costs.

3. The architecture of APERM

3.1 Locality in the memory subsystem

The choice for a regular structure in the architecture necessitates the distribution of storage elements in the design of the parallel reduction machine. However, one does not have completely to abandon the idea of having shared memory. A processor that is reducing a job has to access three types of data-areas, containing respectively the reducer-code, the stacks and the heap. Only the heap-memory contains the reducible expressions and is as a consequence the only part of the storage system that will be involved in the distribution of jobs. An important parameter of a reduction system is the fraction time that a reducer spends in accessing the heap. Measurements on the prototype of APERM have shown that this fraction is only 10% [HAR86]. This means that a shared bus to which the heap-memory is connected will saturate when about 10 processors are simultaneously reducing jobs in the heap-memory. As a consequence an architecture could be based on clusters of about 10 reducers interconnected by a shared bus.

3.2 Communication

In a cluster based concept the inter-cluster communication is of vital importance. If no reducer is active, the highest possible datacommunication performance between adjacent reducer clusters is of the same order as the heap memory bandwidth. This speed could be achieved by coupling two cluster busses with a special processor dedicated to the copying of jobs (e.g. DMA-unit, see figure 3).

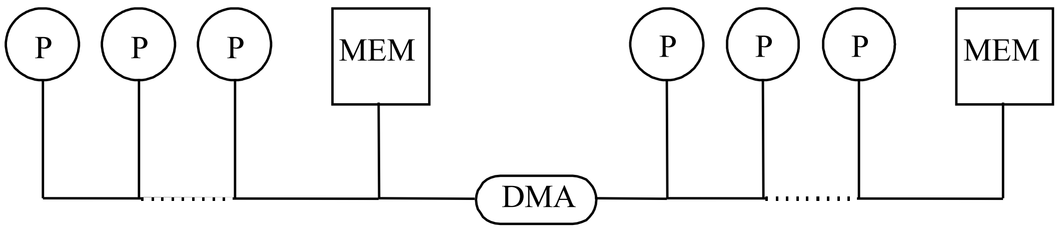


Figure 3: Bus interconnection by DMA

Such a processor acquires mastery over both busses and copies a block from one cluster heap to the other cluster heap. In practice the arbitration protocol on a multi-master bus may cause a considerable overhead. This is caused by the generality of the protocol and the fact that a multi-point arbitration has to be made. Moreover, all processors on both busses are blocked during the DMA transfer.

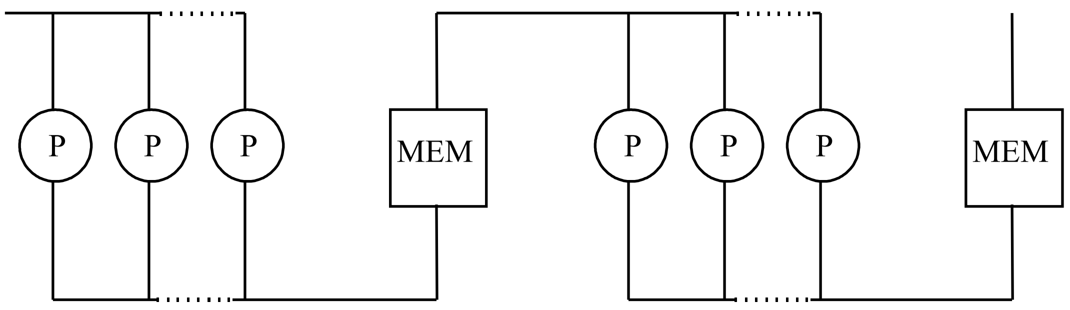


Figure 4: Interconnection by dual ported memory

Another possibility is to equip all memory modules with two busses and fast arbitration logic, resulting in a dual ported memory. The overhead of the arbitration logic on the memory modules can be made sufficiently small to be neglected in comparison with the memory cycle time. This can be achieved because only a two points arbitration has to be made and no general protocol is needed. Figure 4 shows the interconnection between reducer processors that is used in APERM and that is based on dual ported memory modules.

An additional advantage of dual-ported memories is the possibility to avoid data-communication altogether, when a job is allocated to a reducer residing on an adjacent processor. In such cases the destination processor is able to access the job at its original position. There is no need to copy the job-graph, because the sandwich strategy guarantees the job to be a primary redex. This means that all outside references to subgraphs in the job refer to data (normal forms) and reading data can be done by several parallel reducers without conflicts.

3.3 The experimental machine

One of the goals in the development of APERM was to be able to adapt to the progress in VLSI technology. This implies that the machine design had to fulfil the criteria for extensibility, e.g the possibility to add processing power without generating potential bottle-necks.

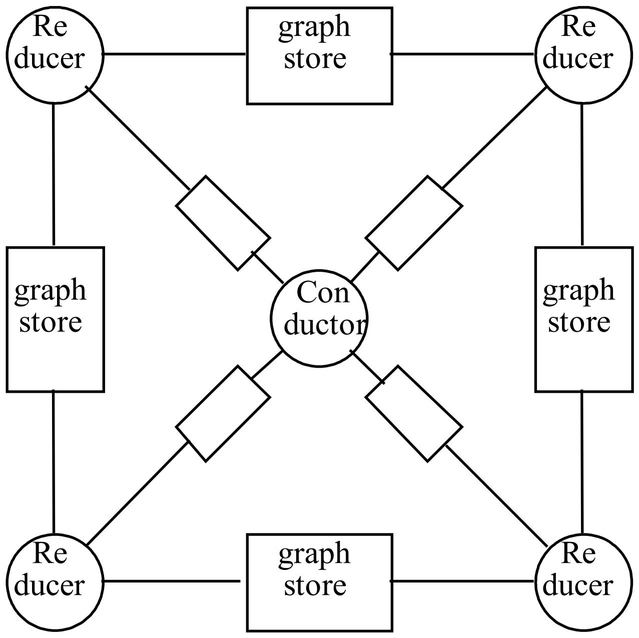


Figure 5: The basic cell

In figure 5 the current prototype of our reduction machine is illustrated. In this prototype one conductor controls the work of four reducer processors and communicates its control information via dual ported memories. The reducer processors are also connected via dual ported memories. These heap-memories are considerably larger than the previous ones because they contain the graph of the program that has to be reduced. When coarse grain reduction tasks have to be transported from one reducer to another this is done via network software that uses the facilities provided by the dual ported memories. In our first prototype a reducer processor prepares a graph for transport whereas another processor reads it from the dual ported memory and transports it to the next memory. This implies that in our current situation the network requires cycles from the reducer processors.

To illustrate the extensibility of the architecture, figure 6 presents a configuration of four basic cells of our machine design. It consists of two separate layers. One layer contains processors responsible for the reduction work and the network for the exchange of coarse grain reduction tasks. The other layer contains the conductors and the network over which they can exchange loadbalancing information. In the example we have assumed a serial connection between the conductors, because we expect that a limited amount of information will be exchanged between the conductors.

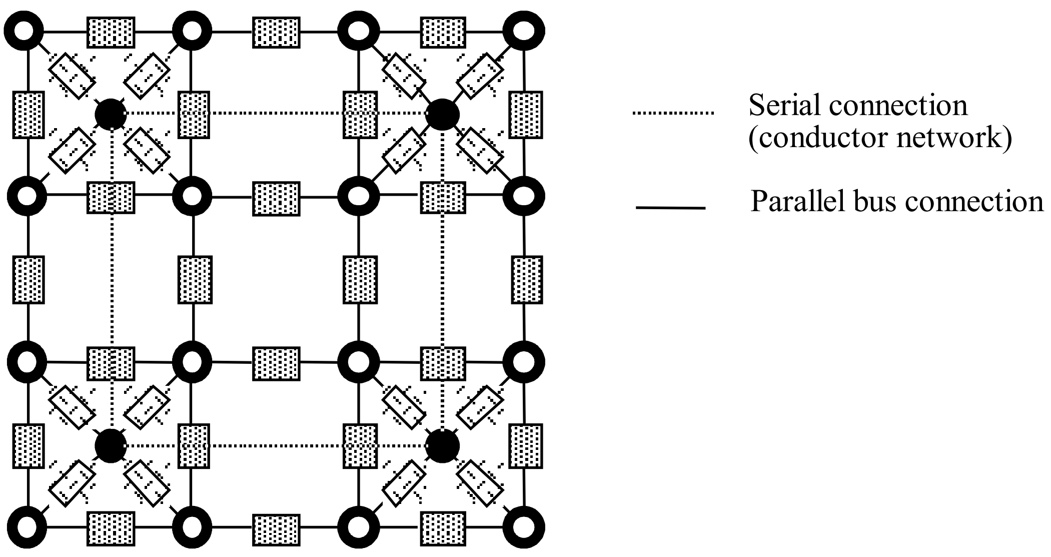


Figure 6: Four basic cells

The reducers are extensible in a mesh type of structure. The conductors, however, can either be extended horizontally or hierarchically, dependent on the flow of information between them.

3.4 Possibilities for VLSI

In section 3.3 it was discussed that in APERM the transport of reducible coarse grain tasks is done via a communication network that uses dual ported memories. The implementation of the current solution of the network layer requires cycles from the processors that also execute the reduction tasks.

This could be avoided if a separate DMA type of processor would be used to do the actual transport. Such a graph transport module could do much more than a simple DMA transfer. It could also perform the algorithm that assembles job graphs into network messages [HAR88]. Before transportation the algorithm has to run through all the pieces of a graph that are scattered over the whole memory and map them into a block of consecutive memory addresses. As this work is also a prerequisite for the garbage collection algorithm that has to operate on the heap memories the graph transport and garbage collection algorithms can be combined to execute in the same processor. A combined graph compaction- and garbage collection processor is our first candidate for VLSI implementation.

4. Performance Measurements

To obtain an impression of the possible performance gain of the job-based reduction model, we have made a partial implementation of the model on our machine architecture. In this partial implementation only one reducer process is active on each processor and communication is only allowed between adjacent processors. With such an experimental set-up transport costs of job graphs can be measured on a point to point basis. The measurements comprise the number of graph-nodes that have been transported for each job and result, the number of reduction steps performed by each job and the actual transmission times of jobs and results. The data obtained are used as parameters in an off-line performance evaluation model of APERM. In this model it is possible to experiment with several loadbalancing strategies and communication optimisations.

In our method to evaluate the performance functional programs on a parallel architecture, we use a mixture of real measurements (communication time, reduction steps) and modeled calculations (loadbalancing, mapping of reduction steps to run-time). We think that this method, which we call *hybrid simulation*, produces accurate performance predictions without incurring the high cost of a full implementation. Only those parts of the architecture are implemented whose simulation would otherwise require an excessive amount of computation time and thus would prohibit a realistic evaluation based on large application programs.

4.1 Results on possible speed-up

A number of application programs has been written in SASL [TUR79], ranging from the fast Fourier transform to a tidal model of the North Sea [VRE87]. Some of the information resulting from a run of an application program on the experimental set-up can be presented as a job graph. In figure 9 such a graph is shown for the Wang-algorithm [WAN81], which eliminates a tri-diagonal set of linear equations. The program contains a cascaded sandwich construct, spawning twice a set of five jobs. In figure 9 the horizontal axis represents execution time, expressed in reduction steps. The horizontal solid lines represent the time that a processor is active reducing a job (bold numbers), whereas the horizontal dotted lines indicate the idle time of a processor. Vertical lines represent the number of nodes that are involved in the transfer of jobs or results (italic numbers).

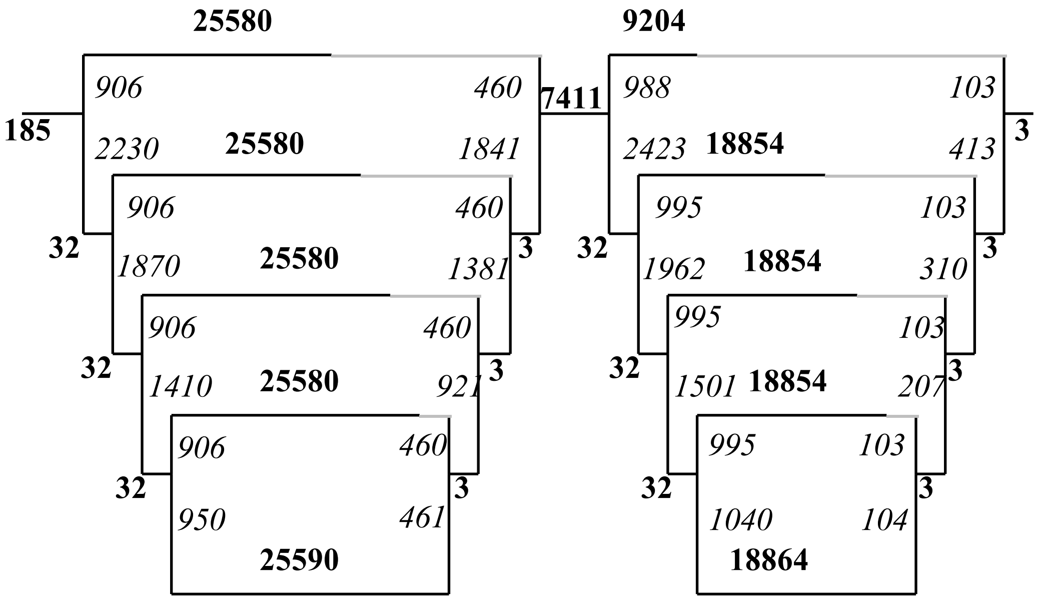


Figure 9: execution profile of the Wang-algorithm on a five processor system

If the architecture provides five or more processors, the job-graph corresponds to an actual execution profile. Otherwise the conductor algorithm has to schedule the jobs on the available processors.

A program that evaluates several variants of the central loadbalancing algorithm is being developed [HOF88]. This analysis program uses the measured point-to-point communication performance and minimises transport cost by exploiting the partly overlapping address spaces of APERM (see next section). Preliminary results indicate that on the fly scheduling heuristics approaches the best possible schedules very closely (within 10%) for all our application runs. However, this centralised heuristic loadbalancing algorithm, only performs slightly better than a simple diffusion scheme.

In section 2.1 we explained that a grain size measure for parallel jobs has to be provided by the application programmer. This grain size measure is compared against a threshold value to decide if a job is still worth being reduced on a remote processor. All our original application programs had to be transformed to provide this grain size measure [VRE88].

Figure 10 shows an example of speed-up figures obtained for a program that calculates the fast Fourier transform of a list of complex numbers. The results are based on optimum schedules, found by exhaustive search through all possibilities with a branch-and-bound algorithm. The calculation of the schedules uses data measured on the experimental machine, like the communication costs for transmitting jobs and results. Garbage collection overhead has not been accounted for (in fact no garbage collection was needed to run a 512-point fast Fourier transform on the experimental machine). In the vertical direction speed-up is plotted against a range of threshold values on the horizontal axis (see figure 10). Each threshold value represents the minimum grain size for parallel jobs in a particular run of the application program. In this example the length of the data list to be transformed is taken as measure for the grain size of jobs. Consequently the threshold value is the minimum length of the data-list required for a job to be executed in parallel.

All speed-up values are calculated with respect to the untransformed sequential version of the program. The fast Fourier transform has been applied to a list of 512 complex numbers. Several speed-up curves have been drawn, each for a different number of available processors.

Going from left to right in figure 10, the number of jobs increases because of a decreasing grain size. As long as processors are under-utilised speed-up improves. However, when too many jobs are generated, speed-up deteriorates because of communication costs. From the figures an optimum threshold value can be determined for each number of processors. For instance in figure 10 the optimum threshold value for a 4 processor system (np=4) is 128. The speed-up of the 512-point Fourier transform is slightly more than 3 in this case.

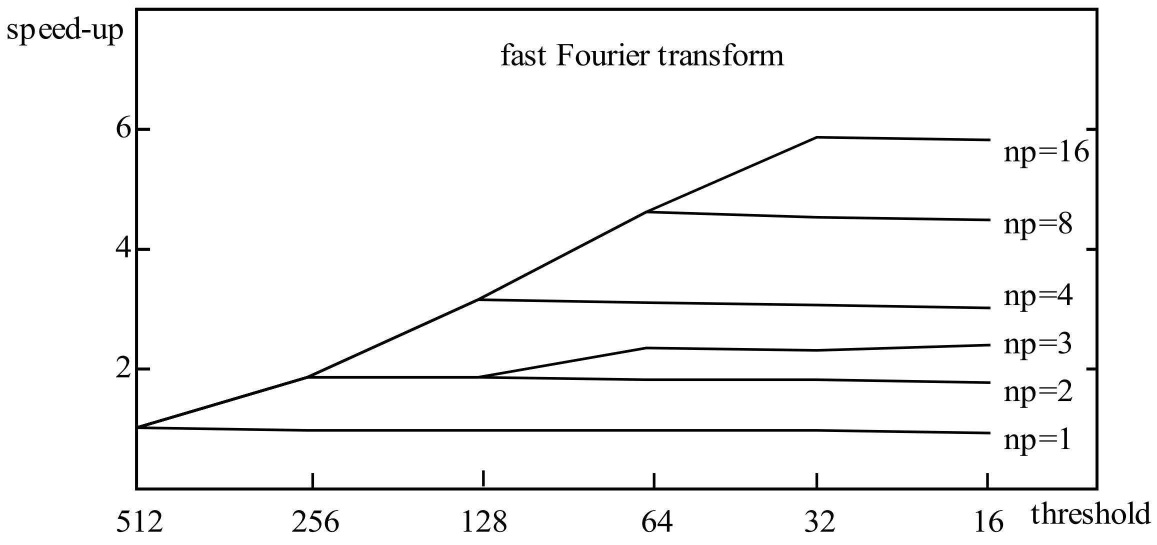


Figure 10: speed-up curves for the fast Fourier transform

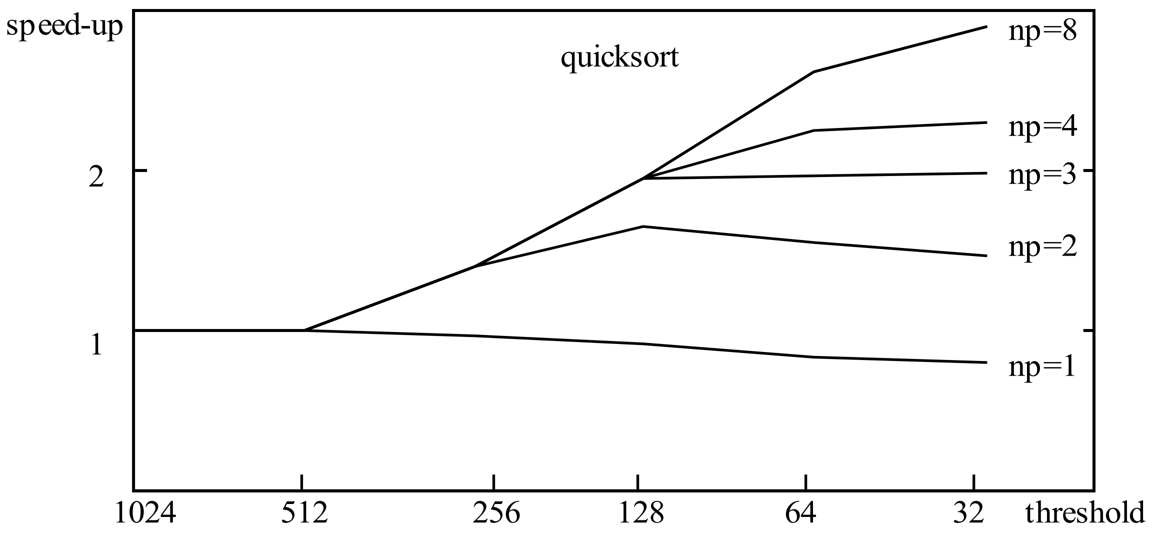


Figure 11: speed-up curves for the quicksort program

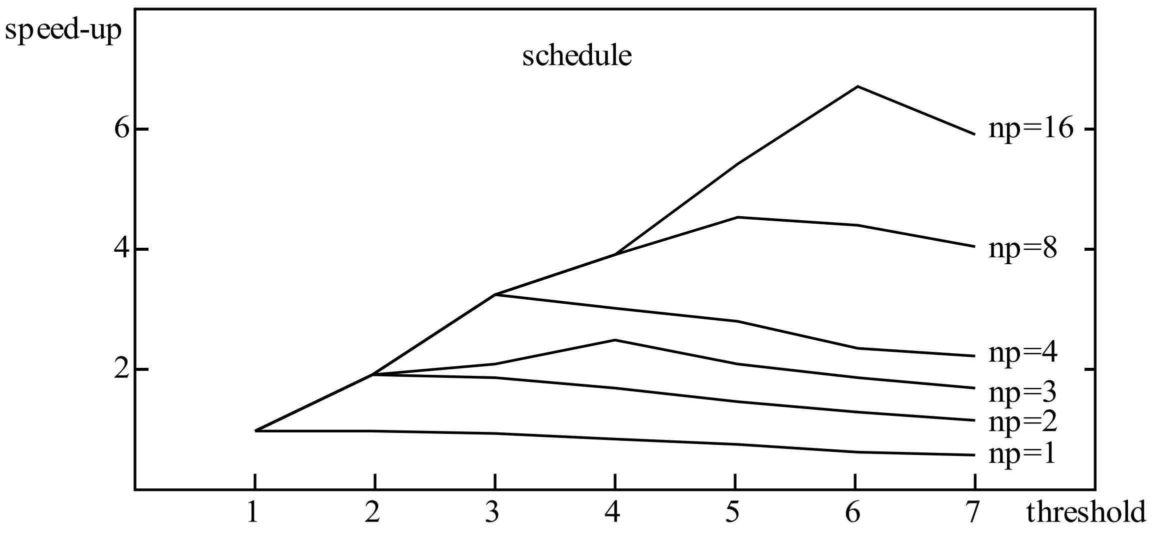


Figure 12: speed-up curves for the schedule program

Similar illustrations (figures 11 and 12) are shown for the quick-sort algorithm and for a parallel version of the branch-and-bound algorithm that we used to calculate the optimum schedules. The threshold value for quick-sort is again the length of the list, but the schedule program uses the depth of the search tree to control the grain size. The input of quicksort is a list of 1024 values obtained by applying the sine function to the numbers 1 to 1024. The schedule program is given a list containing the execution profile of seven hypothetical jobs.

Figure 13 shows the execution profile of the tidal model of the North Sea on a two processor system. Due to the small communication overhead of this program a speed-up of 1.7 is obtained. When the size of the grid in the simulation is *n*, the amount of computation in the parallel jobs grows with $O(n^2)$, whereas the communication cost only grows with $O(n)$.

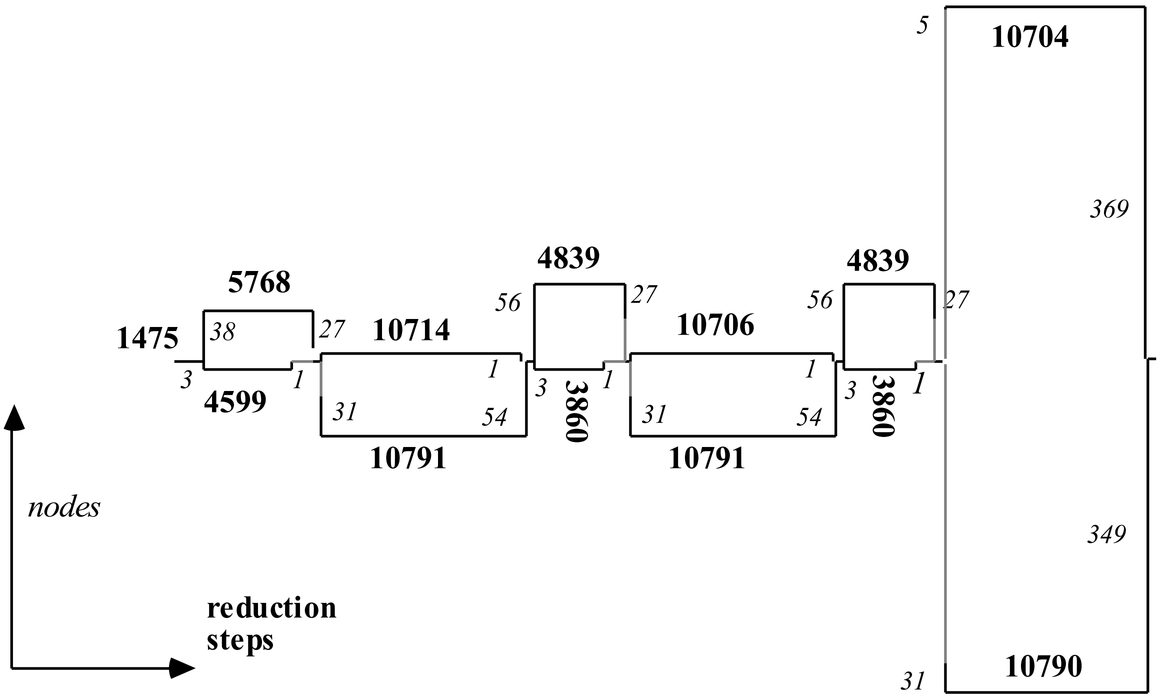


Figure 13: execution profile of the tidal model on a two processor system

4.2 Results on optimising communication

An important aspect of the centralised loadbalancing algorithm is that it tries to minimise the amount of datacommunication. The concrete architecture of APERM (see chapter 4) offers the opportunity to exploit the presence of dual ported heap memories between the reducer processors. The dual ported memories offer a shared memory space between each pair of reducer processors. Preliminary simulation results show that about one third of all jobs that have to be evaluated by a remote reducer need not be copied, because the remote reducer runs in a neighbouring processor and is thus able to access the shared heap space by second port of the memory [HOF88].

For a two processor system the savings are 100%, because both processors are connected to one dual ported memory, and communication is never needed. When the number of processors increases, the economies due to the dual ported memories become less important. This is intuitively clear, because more communication is needed to spread all jobs evenly over the available processors. However, on a sixteen processor system (figure 6), still 33% of the jobs that have to be reduced by a remote processor can avoid communication. We think that these savings together with the large communication bandwidth offered by dual ported memories, justifies the particular architecture of APERM.

5. Comparison and discussion of results

The architecture of APERM differs from a number of related proposals and projects:

- Data communication is based on the use of dual ported memories. Two high-speed parallel busses on the memory modules realise the highest possible datacommunication bandwidth between neighbouring processors. To our knowledge, the only proposal that has some similarity to ours is the Bath concurrent Lisp machine [MAR83], where processors are interconnected by dual ported memories. These memories, however, do not contain the heap storage but merely serve as communication buffers.

Experience shows that in real systems datacommunication represents a major bottle-neck. Most recent and current proposals do not pay enough attention to the implementation of datacommunication. This is due to the fact that often architectures are simulated. If reasonably sized application programmes have to be executed on a machine, the simulation of datacommunication behaviour appears to be practically impossible, because of the amount of computation involved. Therefore, we have developed the method of hybrid simulation (see section 4), which allows us to obtain realistic results based on large application programs.

- The architecture is based on partly overlapping local address spaces provided by the dual ported memories. Both the use of shared busses and dual ported memories enable important optimisations in the transportation of jobs. When a reducer dispatches a job for parallel execution to one of the neighbouring processors, no data communication is needed when the job resides in the overlapping address space. We have shown that in this way a considerable reduction in communication costs can be obtained by the conductor.

- APERM does not support a global address space. This implies that if a job lies outside the local address space of the reducer to which it has been allocated, it has to be copied to this local address space. In case an expression in the functional program is shared by a number of jobs, the copying of this expression can be done without the duplication of work. This requires a special reduction strategy. Many proposals that are based on a local memory architecture still do support a global address space [WAT87, HUD85, KEL84,86]. In our opinion this might introduce overhead that is difficult to control.

- The loadbalancing decisions are made by a logically centralised conductor. The conductor possesses global knowledge of the resource usage in the system and of the grain size of jobs. The loadbalancing algorithm will take advantage of this knowledge to achieve a near optimal distribution of jobs. According to our knowledge all of the recent and current proposals for parallel reduction machines use some form of diffusion scheduling [BUR81, MCB87, KEL84,86]. Preliminary results indicate that our loadbalancing method performs slightly better than diffusion scheduling. Wether this advantage remains for a distributed implementation of the conductor will be a subject for future research.

6. Acknowledgement

We gratefully acknowledge the cooperation with the team members of the PRM project, in particular P.H. Hartel and R.F.H. Hofman.

7. References

[BAR87] H.P. Barendregt, M.C.J.D. van Eekelen, M.J.Plasmeijer, P.H. Hartel, L.O. Hertzberger, W.G. Vree, "The Dutch Parallel Reduction Machine Project", Frontiers in Computing, Amsterdam , December 1987.

[BUR81] F.W. Burton, M.R. Sleep, "Executing functional programs on a virtual tree of processors", Conference on functional languages and computer architecture, New Hampshire, 1981, pp 187-194.

[FAI87] J. Fairbairn, S. Wray, "Tim: a simple, lazy abstract machine to execute supercombinators", Proceedings of the Third International Conference on Functional Programming Languages and Computer Architecture (FPCA '87), Portland , Oregon, USA, LNCS 274, pp. 364-384, September 1987.

[HAR86] P.H. Hartel, W.G. Vree, "A load distribution network for a multi processor reduction machine", Internal report D-6, PRM project, April 1986

[HAR88] P.H. Hartel,W.G. Vree, Parallel graph reduction for divide and conquer applications - part II,Internal report D-20, PRM project, December 1987.

[HOF88] R.F.H. Hofman, "An on-the-fly scheduling algorithm for a parallel reduction machine", Internal report D-18, PRM project, October 1988.

[HUD85] P. Hudak, "Distributed execution of functional programs using serial combinators", IEEE transactions on computers, Vol C-34, number 10, October 1985.

[JOH84] T. Johnsson, "Efficient Compilation of Lazy Evaluation", Proc. of the ACM Sigplan '84, Sigplan Notices, Vol. 19, No 6, June 1984.

[JOH85] T. Johnsson, "Lambda lifting", Proc. Aspenas workshop on implementations of functional languages, Sweden February 1985

[KEL84] R.M. Keller, F.C.H. Lin, "Simulated performance of a reduction based multiprocessor", IEEE Computer, Vol 17, July 1984

[KEL86] R.M. Keller, J.W. Slater, K.V. Likes, "Overview of Rediflow II Development", Proceedings of a workshop on graph reduction, September/October 1986, Santa Fé, New Mexico, USA, LNCS 279, pp 203-214

[MAR83] J. Marti, J. Fitch, "The Bath concurrent Lisp machine", Eurocal, 1983, Lecture notes in computer science 162, pp 78-90

[MCB87] D.L. McBurney, M.R. Sleep, "Transputer-based experiments with the ZAPP architecture", PARLE '87, LNCS 258, pp 242-259

[PEY87] S.L. Peyton-Jones, C. Clack, J. Salkild, M. Hardie, "GRIP-a high performance architecture for parallel graph reduction", Proceedings of the Third International Conference on Functional Programming Languages and Computer Architecture (FPCA '87), Portland , Oregon, USA, LNCS 274, pp. 98-112, September 1987.

[TUR79] D.A.Turner, "A new implementation technique for applicative languages", Software practice and experience, Vol 9, pp 31-49, 1979

[VRE87] W.G. Vree, "The grain size of parallel computations in a functional program", Proc. of the int. conf. on parallel processing and applications, l'Aquila, Italy September 1987.

[VRE88] P.H. Hartel, W.G. Vree,Parallel graph reduction for divide and conquer applications - part I, Internal report D-15, PRM project, December 1987.

[VRE89] W.G. Vree, "Parallel graph reduction for communicating sequential processes", Internal report D-26, PRM project, Februari 1989.

[WAN81] H.H. Wang, "A parallel method for Tridiagonal Equations", ACM transactions on Mathematical Software, Vol 7, No 2, June 1981, pp 170-183

[WAT87] P. Watson, I. Watson, "Evaluating functional program on the FLAGSHIP machine", Proceedings of the Third International Conference on Functional Programming Languages and Computer Architecture (FPCA '87), Portland , Oregon, USA, LNCS 274, pp. 80-97, September 1987.

Chapter V

PARALLEL GRAPH REDUCTION FOR DIVIDE-AND-CONQUER APPLICATIONS PART 1 - PROGRAM TRANSFORMATIONS[1]

[1] PRM project internal report D-15, Dept. of Comp. Sys., Univ. of Amsterdam, December 1988

Parallel graph reduction for divide-and-conquer applications†
Part I - program transformations

Willem G. Vree

Pieter H. Hartel

Computer Systems Department, University of Amsterdam
Kruislaan 409, 1098 SJ Amsterdam

Abstract

A proposal is made to base parallel evaluation of functional programs on graph reduction combined with a form of string reduction that avoids duplication of work. Pure graph reduction poses some rather difficult problems to implement on a parallel reduction machine, but with certain restrictions, parallel evaluation becomes feasible. The restrictions manifest themselves in the class of application programs that may benefit from a speedup due to parallel evaluation. Two transformations are required to obtain a suitable version of such programs for the class of architectures considered. It is conceivable that programming tools can be developed to assist the programmer in applying the transformations, but we have not investigated such possibilities. To demonstrate the viability of the method we present four application programs with a complexity ranging from quick sort to a simulation of the tidal waves in the North sea.

Key words: divide-and-conquer parallel algorithms parallel graph reduction reduction strategy program annotation program transformation job lifting

1. Introduction

Several parallel architectures have been proposed to support the reduction model of computation. These are based on either string reduction[1, 2, 3, 4] or on graph reduction.[5, 6, 7, 8, 9, 10, 11, 12] It is often claimed, that for most application programs, graph reduction is more efficient than string reduction. This is due to the fact, that computational work may be shared; upon completion of the work, the result may be used by all interested parties. In this part of our

† This work is supported by the Dutch Ministry of Science and Education, dienst Wetenschapsbeleid

paper a mixed reduction model based on normal order evaluation is proposed, which shares some of the advantages of both string and graph reduction.

1.1. A storage hierarchy

In graph reduction, a program being executed is represented as a connected graph. Therefore, the graph must be kept in a single storage space. Such a storage space can be implemented in a distributed fashion. In order not to reduce the advantages of sharing, the more frequently non-local accesses occur, the more efficiently they must be performed. In its full generality, this brings about some difficult problems, in particular in the area of garbage collection.[13]

Our basic model of a distributed architecture is that of a communication network, with processing elements at the nodes. Each processing element has its private store. In most implementations of such architectures, the latency of an access to a non-local store is larger by several orders of magnitude than that of a local access. The slow access is usually implemented in software by interprocess communication through a (serial) data-communication network. Fast access to the local stores is based on exactly the same principles, but the implementation details are different. The communication network is usually a fast parallel bus and the interprocess communication occurs between hardware implemented processes of both the memory and the processor. We do not want to dwell on these details but only stress the large difference in speed between local and global access. An implementation should acknowledge this fact by introducing a distinct category of access primitives for global respectively local access.

The purpose of parallel reduction is to speed up computation with respect to sequential reduction. This is achieved by steering the evaluation process in such a way, that reducible expressions appear, which are suitable for evaluation by separate processing elements. The criteria for the selection of such redexes are manifold. For example the granularity of the redexes and their storage requirements play a role. The elected redexes are henceforth called jobs.

In our proposal, programs are annotated via the use of a special primitive function. This provides the mechanism by which jobs are announced at run time. When invoked, the subgraph that represents a job is isolated from the rest of the graph, and made self contained. The subgraph is transferred to the private store of the processing element, which is given the task of normalising the job. Upon completion, the resulting subgraph is merged with the original graph. The previously mentioned global access primitives are used exclusively to implement the transfer of jobs and results. The local access primitives are used to dereference pointers in subgraphs, create new nodes etc.

An important consequence of this evaluation strategy is that application programs must (be made to) exhibit the right kind of locality in space. Otherwise it is inefficient to evaluate jobs in isolation. String reduction provides this locality in a natural way. Therefore we borrow this property by implanting it in a graph reduction system and show that the disadvantages of string reduction can be avoided.

Our attention is devoted mainly to the development of methods by which applications can be made to exhibit locality in space. This has the advantage, that the choice of reduction system can be separated from issues involved with parallelism. In our opinion it does not matter whether a parallel grain is actually evaluated as one reduction step, or as a number of reduction steps. It is far more important that the grain size, the communication cost and the parallel overhead are well balanced. Since the proposed method is not dependent on any particular reduction system, we also benefit from the more practical advantage that our attention is not sidetracked by new developments in the area of fast sequential reduction methods. Since our project was started, three such discoveries were published.[14, 15, 16]

1.2. Applications

Given a particular application, two different methods can be applied to obtain an optimum in the trade-off between the amount of parallelism and the grain size of parallel computations:

Data partitioning

This technique applies when the grain size of an application is too large and can be reduced to produce more and finer grains. Divide-and-conquer algorithms use this technique and are the subject of study in the remainder of this paper. Data partitioning can be summarised as:

F (union (a, b)) → union ((F a) *in parallel with* (F b))

Data grouping

The grouping technique may be applied when the grain size is too small, but an abundant amount of parallelism is available. Several small grains may be combined into one larger grain, as is shown in the following example:

ParMap F (1..10) → SeqMap F (1..5) *in parallel with* SeqMap F (6..10)

Although the example strongly resembles the divide-and-conquer strategy, the mechanism is different. The function *ParMap* is a parallel version of the sequential *map* (apply to all) function, *SeqMap*. In the example *ParMap* distributes each function application ($F\ i$), for $i \in 1..10$ to a different processor, whereas *SeqMap* performs five applications of F in one “grain”.

Not all applications may benefit from parallel evaluation on our system. In particular, if the efficiency of a program is based on sharing, which is the case with for instance the Hamming problem,[17] then we accept that it cannot benefit from parallel evaluation.

In the remainder of the paper we will concentrate on the mechanisms and policies involved in creating and performing “jobs”.

2. Job creation

A multiprocessor architecture without a global store limits the amount of parallelism in a functional program that can be usefully exploited, because the communication cost to transport an expression from one local store to another will often dwarf the gain that is obtained by the parallel reduction of that expression. For this reason we have decided only to allow parallel reduction of certain expressions that comply with the notion of a job. We assume, that initially a single expression is presented for evaluation. There must be a significant amount of work involved in this main expression. A job is defined as a reducible expression with the following properties (the so called job conditions):

1. A job is a closed subexpression (i.e. it contains no free variables).
2. It's normal form is needed† in the main expression.
3. For all concurrent jobs, the communication cost to transport a job must be less than the sum of the reduction costs of the other jobs.

Only subexpressions that are jobs can be submitted to another processor in order to be reduced (in parallel to the main expression and other jobs) by a separate reducer process. It is the responsibility of the programmer to ensure that all job conditions are met. Otherwise parallel evaluation may even cause performance degradation.

The restriction of parallel reduction to jobs bears the following advantages:

- Data communication can be based on jobs (and their results) as the smallest quantity of data to be transported. Communication overhead is small compared to communication cost, since in our proposal not just a single packet is transported,[7, 12] but a complete subgraph.
- Since a job is a closed subexpression, it can be reduced in a separate address space. As a consequence no global garbage collection is needed.
- The process reducing a job is not disturbed by other reducing processes trying to access parts of the job, because all other processes also reduce closed expressions. A reducer only communicates if it needs the result (normal form) of a job submitted by the reducer itself.
- The parallel reduction of a set of jobs starting at the same time is faster than the sequential reduction of these jobs, provided that sufficient processors are available. The problem of achieving a near optimal distribution of jobs over the available processors during run time has to be solved by an additional load balancing mechanism. The calculation of optimal schedules is pursued in part II of this paper.

† A subexpression M is needed in a context $C[M]$ if and only if M is reduced to normal form when $C[M]$ is reduced to normal form.

To prove the last point we need to formalise job condition (3). Suppose there are n jobs with communication cost c_i and reduction cost s_i, $i \in 1 \,..\, n$, where c_i and s_i are measured in the same time unit. Job condition (3) then becomes:

$$\underset{i \in 1..n}{\forall} \left(c_i < \sum_{k=1,\ k \neq i}^{n} s_k \right) \tag{1}$$

What we want to prove is that the longest job (communication included) takes less time than all jobs in sequence (without communication), i.e.:

$$\sum_{k=1}^{n} s_k > \max_{k=1}^{n} (s_k + c_k) \tag{2}$$

From (1) it follows that: $\underset{i \in 1..n}{\forall} \left(c_i + s_i < \sum_{k=1}^{n} s_k \right)$ and therefore (2).

The intuitive version of job condition (3), namely $\underset{i \in 1..n}{\forall} c_i < s_i$ is not sufficient to proof (2). Counter example: two jobs with $c_1 < s_1$, $c_2 < s_2$ and $c_1 > s_2$.

2.1. Sharing

To illustrate the consequences of the job concept for parallel graph reduction we will consider the graphical representation of expressions and rephrase job condition (1):

1. The representation of a job is a subgraph (i.e. there are no references to nodes external to the job).

This condition does not allow for two (or more) jobs to share a subgraph. In the illustration of figure (1) graphs A and B share the subgraph C. Therefore, graph A does not qualify as a job because it contains an external pointer to C.

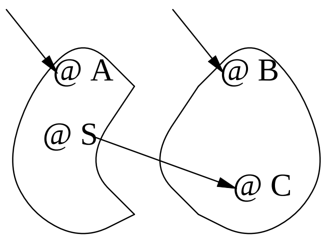


Figure 1 : An external pointer

There are several reasons not to extend the definition of a job to support these external pointers:

- Before submitting a job (B) all sharing nodes (such as S) have to be discovered and flagged. This is necessary because otherwise the process trying to reduce a sharing node (S) would not know where to find the result (C). The discovery of sharing nodes is a time consuming process because the whole graph has to be traversed and marked.

- The amount of work to reduce a shared expression might be small.
- After the reduction of job *B* it is not certain that the expression *C* has also been reduced. This is the case for example if *C* is not needed in expression *B* (e.g. $B =$ if "*true*" then $\cdots$ else C). So *A* might have waited for a result and still have to do the work.

Considering these difficulties we have decided not to support sharing between jobs and to keep jobs completely self contained. This implies that sharing may only occur within a job. In the example of figure (1) it means that before sending away job *B* the subexpression *C* is copied, and both jobs *A* and *B* will reduce *C*.

2.2. Duplication of work

The performance gain attained by parallel reduction might well be cancelled by the duplication of work inherent to ordinary string reduction, as is shown in the illustration of figure (2).

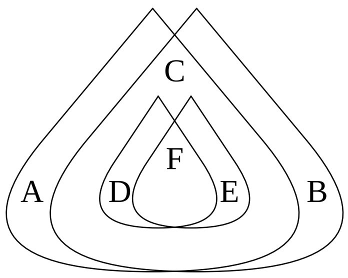


Figure 2 : Nested sharing

The job *C* is reduced twice, once as part of job *A* and once as part of job *B*. However, since *D* and *E* are contained in *C* and share *F*, *F* is computed twice for *C* and thus four times for *A*. The solution is to reduce *F* first, supply its normal form to *D* and *E* and then reduce *C* etc. A special parallel reduction strategy has been designed (the "sandwich"-strategy) that avoids duplication of work. It is demonstrated with practical examples that divide-and-conquer algorithms can be converted into sandwich programs.

2.3. The sandwich strategy

In a system that exploits strict operator parallelism, a simple job administration is all that is necessary. For example, if some reduction sequence encounters the redex (*TRIPLUS* x y z), the addition can not be performed until all arguments have been normalised (in parallel). Hence there is no need for the job corresponding to argument x to reactivate the addition before jobs y and z have completed or vice versa.

In contrast to strict operator parallelism, a general parallel reduction strategy would allow for any subexpression to be treated as a job. Although more flexible, this has the disadvantage that the administration of jobs is more complex. Suppose, that the generation of parallelism is triggered by annotating subexpressions. For the application cited above there are several possible ways to annotate one or more of the three arguments. Any completely normalised argument will cause the addition to be reactivated, with the chance that no further progress can be made because some of the arguments are still unavailable. The sandwich strategy combines the advantages of the simple job administration required for strict operator parallelism and the possibility to annotate arbitrary subexpressions, at the detriment of some flexibility.

A sandwich expression is defined as a needed function application $(G\ x_1\ x_2 \cdots x_n)$ with the following restrictions (the sandwich conditions):

1. The function G is strict† in all argument positions.
2. Each argument x_i of G is a function application $(H_i\ a_{i1}\ a_{i2} \cdots a_{ik_i})$ where:
3. The function H_i is strict in all its arguments.
4. Each expression $(H_i\ a_{i1}\ a_{i2} \cdots a_{ik_i})$ satisfies the job conditions.
5. The expressions H_i an a_{ij} are in normal form.

Given a sandwich-expression, the sandwich strategy now runs as follows:

- Submit all function applications $(H_i\ a_{i1}\ a_{i2} \cdots a_{ik_i})$ as separate jobs to be reduced in parallel.
- Wait for the results of all submitted jobs and continue with the normal order reduction of G, applied to the results just received.

The sandwich strategy never duplicates work, because when jobs are submitted and copying takes place, all terms in question are in normal form (H_i and a_{ij}). Thus only normal forms are copied and these, by definition, do not contain work. The strategy has been named a “sandwich” because it consists of one layer of parallel and applicative evaluation between two layers of normal evaluation.

In the framework of the SASL programming language[18] a new primitive function has been introduced, which implements the sandwich strategy. The general form of a sandwich application is:

$$\textit{sandwich}\ \ G\ \ (H_1\ \ a_{11} \cdots a_{1k_1}\ ) \cdots (H_n\ \ a_{n1} \cdots a_{nk_n}\ )$$

Apart from parallel evaluation, the expression is equivalent to:

$$G\ \ (H_1\ \ a_{11} \cdots a_{1k_1}\ ) \cdots (H_n\ \ a_{n1} \cdots a_{nk_n}\ )$$

The *sandwich* function evaluates the applications $(H_i\ \ a_{i1} \cdots a_{ik_i})$ in parallel. As soon as the results of these evaluations have become available, normal lazy evaluation resumes.

† A function G with arity n is strict in argument position i if for every possible redex R, R is needed in $(G\ x_1 \cdots x_{i-1}\ R\ x_{i+1} \cdots x_n)$.

Summarising, we propose to perform graph reduction within a job and string reduction without duplication of work on the parallel job level. The sandwich strategy exploits strict operator parallelism, but allows the programmer to define the operator.

3. Job control

The sandwich strategy provides the means to generate an abundant amount of parallelism, since jobs may contain sandwich expressions, which create new jobs etc.

There are two points worth noting:

- Since we strive at obtaining best results with divide-and-conquer problems, it may be assumed that creating more jobs implies that the individual jobs become smaller (in terms of computational work), up to a point where job condition (3) no longer applies.
- For large problems, an uncontrolled expansion of the population of jobs will outgrow even the most powerful architecture.

Some form of "job control" is necessary to prevent the system from being flooded with small jobs. A good control mechanism would not unduly restrain parallelism, because idle processing elements are a waste of resources. In general the control mechanism must be adaptive to the load of the system.

In the architecture proposed here, there is no need for an application independent control mechanism, since all divide-and-conquer algorithms provide a "handle" for regulating the generation of jobs. It is sufficient to make the parallel divide phase conditional to the grain size of the potential jobs. A consequence of the relation between the amount of work involved in the individual jobs and their number is, that a mechanism aimed at keeping the grain size large enough will automatically restrain the number of jobs. A threshold on the grain size is necessary and sufficient.

In the next sections examples are given of how the grain size of jobs in divide-and-conquer problems can be calculated and controlled at source level via program transformation. In most other proposals, this control is exerted at a lower level,[10, 19, 20] which makes it harder to devise good heuristics.

4. Application of the sandwich strategy

As a first example of transforming a divide-and-conquer algorithm into a version suitable for parallel evaluation we consider the quick sort algorithm. The principle of our transformation also applies to other divide-and-conquer algorithms, as will be shown by a parallel version of the fast Fourier transform, Wang's algorithm for solving a sparse system of linear equations and a hydraulical simulation program.

4.1. Quick sort

Figure (3) shows the quick sort algorithm in SASL.[21] Before proceeding we will briefly introduce the aspects of the SASL programming language that we need here. A function definition in SASL may consist of several equations. For instance in figure (3) there is one definition of *QuickSort*, with two alternatives selected by pattern matching on the actual argument. If the list to be sorted is empty, which is written as ( ), the empty list is produced as the answer. In the second equation, the formal argument to *QuickSort* specifies a pattern $(a: x)$, which is matched with the actual argument when *QuickSort* is applied to a non-empty list. During the pattern match the head and the tail of the actual argument are made accessible as a and x respectively. In the WHERE definition something similar happens. The result of the application $Split\ a\ x$ ( ) ( ) must be a list, the head of which is made accessible as m and the tail as n.

When applied to a non-empty list, *QuickSort* selects the head a of the list and supplies this element as the pivot to the function *Split*. The tail x of the input list is split around the pivot in two sublists m and n. These sublists are subsequently supplied to recursive invocations of *QuickSort*. The symbol (++) denotes the infix operator that appends the right list argument to the left one. When (:) is used as an operator in an expression it prepends a new head to a list.

In the *Split* function a conditional is used to collect the list elements with a value lower than the pivot in the accumulating argument m. The remaining list elements are collected in the second accumulating argument n. The arrow (→) connecting a condition and a clause should be read as *then*. The else clause of the conditional is $Split\ a\ y\ m\ (b: n)$.

$$\begin{aligned}
QuickSort\ (\,) &= (\,)\\
QuickSort\ (a: x) &= (QuickSort\ m) + + (a: (QuickSort\ n))\\
&\ \text{WHERE}\\
&\ m: n = Split\ a\ x\ (\,)\ (\,)\\[6pt]
Split\ a\ (\,)\ m\ n &= m: n\\
Split\ a\ (b: y)\ m\ n &= b < a \rightarrow Split\ a\ y\ (b: m)\ n\\
&\ Split\ a\ y\ m\ (b: n)
\end{aligned}$$

Figure 3 : Sequential quick sort application

To obtain a parallel version of a program, subexpressions that can be evaluated in parallel must be annotated. To achieve this we use angular brackets (‹ and ›), which obey the same syntactic rules as the normal parentheses. An expression between matching angular brackets is a job. Figure (4) shows the version of the *QuickSort* function after annotation with job brackets. The annotation has to be provided by the programmer.

$$
\begin{aligned}
\mathit{QuickSort}\ (\,) &= (\,)\\
\mathit{QuickSort}\ (a\!:x) &= \ \langle\, \mathit{QuickSort}\ m\,\rangle\ +\!+\ (a\!:\ \langle\, \mathit{QuickSort}\ n\,\rangle\,)\\
&\ \text{WHERE}\\
&\ m\!:n = \mathit{Split}\ a\ x\ (\,)\ (\,)
\end{aligned}
$$

Figure 4 : Quick sort annotated by the programmer with job brackets

A program annotated with job brackets can be transformed more or less automatically into a version with sandwich expressions. A formal description of the transformation may be found in chapter 6. In the remainder of this chapter we will introduce the principles of the transformation by means of a series of examples.

The transformation requires two steps. The first step, which we call job lifting, recognises expressions between job brackets. Job lifting generates an auxiliary function G that satisfies the sandwich conditions. In figure (5) job lifting has replaced the body of *QuickSort* by a sandwich expression of G.

$$
\begin{aligned}
\mathit{QuickSort}\ (\,) &= (\,)\\
\mathit{QuickSort}\ (a : x) &= \mathit{sandwich}'\ G\ (\mathit{QuickSort}\ m)\ (\mathit{QuickSort}\ n)\\
&\ \text{WHERE}\\
&\ G\ P\ Q = P +\!+ (a : Q)\\
&\ m : n = \mathit{Split}\ a\ x\ (\,)\ (\,)
\end{aligned}
$$

Figure 5 : The job lifted version of quick sort

If both applications of *QuickSort* in figure (5) were to be reduced in parallel, the application (*Split a x* ( ) ( )) would be copied and reduced twice. To solve this problem, we introduce a variant *sandwich′* of the *sandwich* primitive, which normalises all the a_{ij} (in casu m and n) before jobs are created. This has the effect of normalising the application (*Split a x* ( ) ( )) before the creation of the jobs.

For the sandwich strategy to be effective, both recursive applications of *QuickSort* in figure (5) should contain enough work to outweigh their communication cost (job condition 3). This may be achieved by imposing a lower limit on the length of the lists m and n. Figure (6) shows the final version of the *QuickSort* program, with controlled application of the sandwich strategy as obtained by a second transformation step. We call this step the grain size transformation. The length of the list to be sorted is taken as a measure of the grain size, since the amount of work is $O\,(\mathit{length}^{\,2} \log \mathit{length})$.

The normalisation forced by the variant *sandwich′* is no longer necessary. The reason is, that to determine the lengths of the sublists m and n, both will have to be normalised. The comparisons to the *Threshold* therefore serve a dual purpose: controlling the grain size and forcing normalisation. Although the final version of the quick sort program has a complex appearance, it should be noted that most of the code is generated by two program transformation steps.

```
              $Threshold = 100$

          $QuickSort\ () = ()$
       $QuickSort\ (a : x) = length\ m > Threshold \rightarrow$
                                 $length\ n > Threshold \rightarrow$
                                       $sandwich\ G\ (QuickSort\ m)\ (QuickSort\ n)$
                                 $QuickSort\ m ++ (a : QuickSort_{seq}\ n)$
                             $length\ n > Threshold \rightarrow QuickSort_{seq}\ m ++ (a : QuickSort\ n)$
                             $QuickSort_{seq}\ m ++ (a : QuickSort_{seq}\ n)$
                             WHERE
                             $G\ P\ Q = P ++ (a : Q)$
                             $m : n = Split\ a\ x\ ()\ ()$
```

....

```
       $QuickSort_{seq}\ () = ()$
    $QuickSort_{seq}\ (a : x) = QuickSort_{seq}\ m ++ (a : QuickSort_{seq}\ n)$
                             WHERE
                             $m : n = Split\ a\ x\ ()\ ()$

         $Split\ a\ ()\ m\ n = m : n$
     $Split\ a\ (b : y)\ m\ n = b < a \rightarrow Split\ a\ y\ (b : m)\ n$
                             $Split\ a\ y\ m\ (b : n)$
```

Figure 6 : Final parallel version of the quick sort program.

The cost involved in the control mechanism that is introduced by the grain size transformation has to be weighed against the benefits from parallel evaluation. The optimal value of the *Threshold* depends on properties of the system configuration. Both issues are pursued in part II of this paper.[22]

4.2. The fast Fourier transform

The fast Fourier transform processor is an early example of parallel computer architecture. Though several different organisations have been proposed for these special purpose processors,[23] none of them exploited the divide-and-conquer strategy to obtain parallelism, because the divide-and-conquer strategy requires many processors executing the same algorithm and processors used to be an expensive resource.

Unlike the quick sort algorithm the fast Fourier transform perfectly divides the data into two equal parts at each recursive invocation. This should allow for an optimal processor utilisation. Using a free mixture of conventional mathematical notation and SASL syntax, the essential part of the program with the job annotation is shown in figure (7).

```
$fft\ 1\ r\ \vec{d} = \vec{d}$
$fft\ n\ r\ \vec{d} = \langle\ fft\ halfn\ halfr\ \vec{u}\ \rangle\ ++ \langle\ fft\ halfn\ (halfr + 128)\ \vec{v}\ \rangle$
        WHERE
        $halfn = n\ /\ 2$
        $halfr = r\ /\ 2$
        $\vec{u} = \vec{x} + \vec{z}$
        $\vec{v} = \vec{x} - \vec{z}$
        $\vec{x}\ , \vec{y} = split\ \vec{d}\ halfn$
        $\vec{z} = \vec{y} * \exp\ (halfr * i * \pi\ /\ 128)$
```

Figure 7 : The annotated 512-point fast Fourier transform program

To simplify the presentation, the length of the data-list to be transformed has been fixed to 512 elements, which explains the origin of the constant 128 in the program. Furthermore the result list produced by this program is not in the right order and has to be passed through a reorder function, which is, again for the sake of simplicity, not shown. For a fixed length fast Fourier transform, like the one in figure (7), the reorder function can be replaced by a fixed mapping. The function application ($split\ \vec{d}\ n$) produces a pair of lists of which the first one contains the first n elements of $\vec{d}$ and the second one contains the rest of $\vec{d}$ (again n elements). The function application (fft 512 0 $\vec{d}$) performs a 512-point fast Fourier transform on the list $\vec{d}$ that contains 512 complex numbers. All arithmetic on the vector variables is assumed to be complex. A vector of complex numbers is represented by a list of pairs, where each pair contains a real and an imaginary part.

Since the *fft* function already requires the length of the list of data as a parameter this information is readily available for the purpose of controlling the grain size. The transformation from the version of the program shown in figure (7) to the final sandwich version with threshold control can be performed according to the guidelines of chapter 6.

4.3. Wang's algorithm for solving a sparse system of linear equations

Many mathematical models of physical reality consist of a set of partial differential equations. An important step in approximating the solution of such a set of equations is to solve a large set of linear equations. The corresponding matrices often appear to be in a tri-diagonal or block tri-diagonal form. Wang has proposed a partitioning algorithm to achieve parallelism in the elimination process of a tri-diagonal system.[24] According to Michielse and van der Vorst[25] a slightly modified algorithm is well suited for local memory parallel architectures. The basic idea of the algorithm is to divide a tri-diagonal matrix in equally sized blocks and to try elimination of these blocks in parallel. The two edge blocks (top left and bottom right) are extended by a zero column, to obtain the same size as the other blocks. Figure (8) shows how a 12×12 matrix can be split into three blocks.

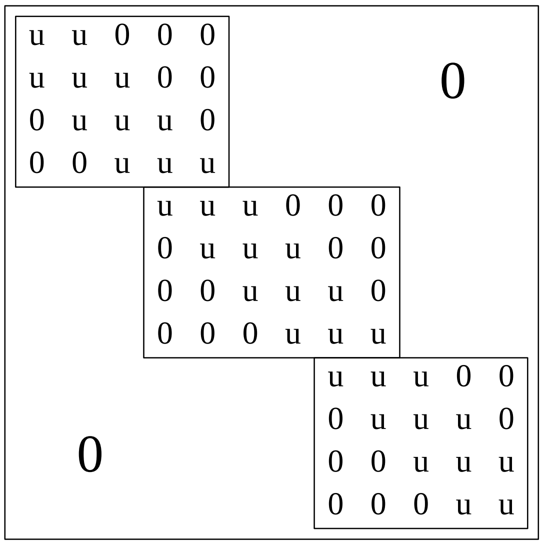


Figure 8 : Partitioning of a tri-diagonal matrix ($u \neq 0$)

Each block can now be eliminated in parallel. Figure (9) illustrates the effect of this part of the algorithm on one block (i.c. the centre block of figure 8).

$$\boxed{\begin{matrix} u & u & u & 0 & 0 & 0 \\ 0 & u & u & u & 0 & 0 \\ 0 & 0 & u & u & u & 0 \\ 0 & 0 & 0 & u & u & u \end{matrix}} \longrightarrow \boxed{\begin{matrix} v & v & 0 & 0 & f & 0 \\ f & 0 & v & 0 & f & 0 \\ f & 0 & 0 & v & f & 0 \\ f & 0 & 0 & 0 & v & u \end{matrix}}$$

Figure 9 : First elimination in one block

The elimination algorithm is designed in such a way, that the fill-in that arises (shown by the letter *f* in figure 9) is confined to the first and fifth columns of the partition. The reason for this confinement becomes apparent when two adjacent blocks that have been processed are shown together, like blocks *A* and *B* in figure (10).

$$
\begin{array}{r|cccccc|cccc}
\cline{2-7}
 & v & v & 0 & 0 & f & 0 & & & & \\
\text{A:} & f & 0 & v & 0 & f & 0 & & & & \\
 & f & 0 & 0 & v & f & 0 & & & & \\
 & f & 0 & 0 & 0 & v & u & & & & \\
\cline{2-11}
 & & & & & v & v & 0 & 0 & f & 0 \\
\text{B:} & & & & & f & 0 & v & 0 & f & 0 \\
 & & & & & f & 0 & 0 & v & f & 0 \\
 & & & & & f & 0 & 0 & 0 & v & u \\
\cline{6-11}
\end{array}
\longrightarrow
\quad f\ \ 0\ \ 0\ \ 0\ \ w\ \ 0\ \ 0\ \ 0\ \ f\ \ 0
$$

Figure 10 : Elimination at the borders of the blocks

The rightmost column containing the fill-in of matrix *A* is the same column as the leftmost column of matrix *B*, which also contains fill-in. When the top row of block *B* is used to eliminate the right most value (*u*) at the bottom row of block *A*, the latter row only contains non-zero values at the row positions where fill-in still has to be eliminated (see the result in figure 10). If the same elimination is performed on all pairs of border rows of adjacent blocks, the resulting bottom rows of all blocks together constitute a tri-diagonal matrix. Figure (11) shows this subsystem for the example matrix and the result of the elimination. This can be achieved either directly with Gauss elimination or if the system is large enough by recursive application of the partitioning algorithm.

$$
\boxed{\begin{array}{ccccc} 0 & w & f & 0 & 0 \\ 0 & f & w & f & 0 \\ 0 & 0 & f & w & 0 \end{array}}
\longrightarrow
\boxed{\begin{array}{ccccc} 0 & x & 0 & 0 & 0 \\ 0 & 0 & x & 0 & 0 \\ 0 & 0 & 0 & x & 0 \end{array}}
$$

Figure 11 : Elimination of the subsystem

After restoring the rows of the solved subsystem into their original positions as bottom rows of each block (see the left matrix in figure 12) it can be observed, that it is possible to eliminate all the fill-in of a block locally, only using the bottom row of the next higher block. This final elimination step is shown in figure (12) and again all blocks can be processed in parallel.

$$\begin{array}{cc|ccccc|} 0 & \dots & x & 0 & 0 & 0 & 0 \\ \cline{3-7} & & v & v & 0 & 0 & f \\ & & f & 0 & v & 0 & f \\ & & f & 0 & 0 & v & f \\ & & f & 0 & 0 & 0 & x \\ \cline{3-7} \end{array} \longrightarrow \boxed{\begin{array}{ccccc} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{array}}$$

Figure 12 : Final elimination

The SASL program that implements the algorithm is shown in figure (13).

```
 Partition matrix = ParMap SecondElimination matrix₂
                    WHERE
                    matrix₂ = SequentialPart matrix₁
                    matrix₁ = ParMap FirstElimination matrix
ParMap f (a : ()) = (f a) : ()
 ParMap f (a : x) = ‹ f a › : ‹ ParMap f x ›
```

Figure 13 : Skeleton of Wang’s algorithm in SASL with annotation

The function *FirstElimination* incorporates the first local block elimination, which is shown in figure (9). The results of this first parallel step are gathered into $matrix_1$, which is subsequently reduced sequentially to $matrix_2$ by the function *SequentialPart*. The latter implements the pair-wise border row elimination of figure (10) and the Gauss elimination of bottom rows from figure (11). Finally, the second local block elimination, which is shown in figure (12), is performed by the function *SecondElimination*.

Parallelism is enforced by the function *ParMap*, which assumes its second argument to be a list. In order for *ParMap* to yield a correct result, the *matrix* should be structured as a list of blocks: ($block_1$, $block_2$, $block_3$, $\cdots$, $block_n$). This list structure does not cause a performance penalty, because it is traversed in a linear sequence by *ParMap*. The grain-size of the parallel computations of this program is completely determined by the size of the blocks into which the matrix is initially divided. In contrast to the previous examples, there is no need for dynamic grain size control (see also figure 24).

5. An extension of the reduction model to support persistent results

The sandwich strategy imposes a restriction on the type of applications that may be alleviated without loosing the advantages of the strategy. For instance during the first phase of the computation in Wang’s algorithm, each job assigned to process a diagonal block of the matrix produces “fill in”, which must be eliminated during the third phase. The values needed for this elimination are calculated in a second phase. The Gauss elimination in that phase only requires the values of the matrix elements in the bottom rows of the matrix blocks. The remaining matrix elements are returned with the results of the first phase, only to be incorporated in new jobs when the third phase is started. So a large part of the matrix is transported twice: once as result of the first phase and once as part of a job in the third phase. The structure of the computations in the third phase is the same as that of the first phase, hence the matrix blocks will probable arrive at the same reducer as before. It would have been more efficient to keep the blocks in their respective places and connect the jobs generated during phase three to the "persistent" blocks.

A mechanism is proposed, by which a subexpression of a result can be marked, with the following interpretation:

- The marked subexpression in a result is replaced by a "remote name" when the result is returned to its creator. Instead of the subexpression, only the remote name is transmitted.
- After transmission of the result, the marked subexpression is saved, with its remote name, for future use on the current reducer.
- When a remote name appears in a job, it will be allocated to the reducer that contains the corresponding (marked) subexpression such that they may be combined to form a complete job. The marking is then automatically destroyed.

A remote name is a unique identification of a subexpression. Except that it is generated and destroyed during reduction, a remote name is similar to the names that may be given to expressions in functional programs. A potential job must not contain more than one remote name, since these may be bound to different physical locations. Outside a job a remote name has no meaning. Furthermore, it may never be dispensed with explicitly, since this would leave an otherwise unreachable subexpression behind, which can not be garbage collected.

5.1. The sandwich and own functions

The primitive function *own* generates a remote name and causes its argument to become a marked subexpression; otherwise it has the same semantics as the identity function. It is sufficient to mark just the root of the graph that represents the subexpression. A remote name is recognised by the *sandwich* function, if it appears as one of the H_i or a_{ij} in its second argument. The restriction to certain positions has the advantage, that the implementation of the *sandwich* function does not have to search for remote names throughout the graph that represents its second argument.

In the example shown in figure (14) the *own* function marks the head of the result list, which is returned by the function H. The latter reuses the value of *newhead* during its next application.

```
repeat oldhead oldtail 1
            = oldhead : oldtail
repeat oldhead oldtail n
            = repeat newhead newtail newn
              WHERE
              newn = n − 1
              newhead : newtail = sandwich′ G (remote oldhead oldtail newn)

remote oldhead oldtail n
            = n = 1 → newhead : newtail
              (own newhead) : newtail
              WHERE
              newhead : newtail = H oldhead oldtail

      H a x = (a + 10) : (x + 7)
    G (a : x) = a : (x + x)
```

Figure 14 : Cooperation of the *sandwich* and *own* functions

To clarify the operational semantics of the *own* and *sandwich* primitives, a number of reductions will be shown that appear during the evaluation of the application (*repeat* 0 0 3). There are two processes involved in this reduction sequence. These have been named *parent* and *child*. The steps carried out by the *child* process are shown offset to the right in figure (15).

step	parent process	step	child process
1	*repeat* 0 0 3		
2	*sandwich′ G* (*remote* 0 0 2)		
		3	*remote* 0 0 2
		4	*H* 0 0
		5	(*own* 10) : 7
6	*G* (“*remote name*” : 7)		
7	*repeat* “*remote name*” 14 2		
8	*sandwich′ G* (*remote* “*remote name*” 14 1)		
		9	*remote* 10 14 1
		10	*H* 10 14
11	*G* (20 : 21)		

Figure 15 : The evaluation of (*repeat* 0 0 3)

The first application of the *sandwich′* function (step 2) is a normal sandwich expression. It creates a job, which is evaluated by the child process. The “remote name” is generated by the application of the *own* function in step 5. It is returned with the result, while the value 10, which it represents is left behind. Via the application of *G* (step 6), The remote name is passed to the next invocation of *repeat* (step 7). The second *sandwich* application (step 8) generates a new job, which carries the remote name back to the child process, where it is replaced by the subexpression 10. By then, the third parameter to the function *remote* has the value 1, such that instead of a (new) remote name, the value 20 is returned with the result. The computation is finished when *G* has produced its result.

In the implementation of this mechanism no global name directory is required, because a remote name carries a system wide address of the expression it represents. This address can be used to send a job containing a remote name from anywhere in the system back to the creator of the remote name.

5.2. A parallel hydraulical simulation

A functional program that implements a mathematical model of the tides in the North Sea[26] has been transformed into a version that will run efficiently on a parallel local memory architecture by the use of the *own* function in combination with the sandwich strategy. To be able to apply the *sandwich* function, the original program, which contains cycles, has to be transformed into a program without cycles. Details of this transformation can be found a paper by one of the authors.[27] Here only the essential skeleton of the program will be used to clarify the annotations.

Without the use of the *own* function the tidal model would retransmit large matrices on each iteration of its main recursion. Consequently the program would run much less efficient on a parallel local memory architecture. The Wang partition algorithm, presented in section 4.3, only suffers a small loss in efficiency without the own-annotation, due to the fact that the matrix blocks are only retransmitted once during the whole calculation.

The physical model of the tides repeatedly updates a matrix that contains approximations of the x-velocity, the y-velocity and the wave height of the water in each point of a spatial grid. In a parallel version of the program the matrix can be split into as many blocks as the degree of parallelism requires. We only present a partitioning of the matrix into two blocks, to concentrate on the annotation issues. Figure (16) shows the main recursion of the program, which is started with two partitions called *Left* and *Right*. These partitions will be updated in parallel.

$$main\ Left\ Right\ n = repeat\ Update_1\ (Left : (Right : (LeftBorderOf\ Right)))\ n$$

$$\begin{aligned} &repeat\ f\ x\ 0 = GetRemoteData\ x \\ &repeat\ f\ x\ n = repeat\ f\ (f\ x)\ (n-1) \end{aligned}$$

Figure 16 : The main recursion of the tidal model

The function $Update_1$ submits the matrices *Left* and *Right* to different processors, where the actual updating takes place in parallel. All subsequent recursive invocations of $Update_1$ will only transmit remote names instead of real matrices, due to the application of the *own* function in the remote processors (see below). Therefore a special function *GetRemoteData* is provided, to force the transmission of the actual matrices at the end of the main recursion. Figure (17) presents the function $Update_1$. The process of the updating itself is split into two phases, after each of which communication of one border of the matrices takes place. The first phase updates the x-velocity in both matrices and is implemented by the functions *UpdateXleft* and *UpdateXright*. In the second phase both the y-velocity and the wave height are updated by the functions *UpdateYHleft* and *UpdateYHright*. Both update phases are dependent on each other and have to be run in sequential order. The left and right parts of each update phase are executed in parallel.

```
Update₁ M = ‹ UpdateYHleft Left₁ › : ‹ UpdateYHright Right₁ BorderOfLeft₁ ›
                WHERE
                (Left₁ : BorderOfLeft₁ ) : Right₁ = Update₂ M

Update₂ (Left₂ : (Right₂ : BorderOfRight₂))
              = ‹ UpdateXleft Left₂ BorderOfRight₂ › : ‹ UpdateXright Right₂ ›
```

Figure 17 : The two phases of the updating with annotations

The illustration of figure (18) shows the desired communication structure of $Update_1$ and $Update_2$. The dashed arrows represent the transmission of remote names, whereas the solid arrows denote communication of real data.

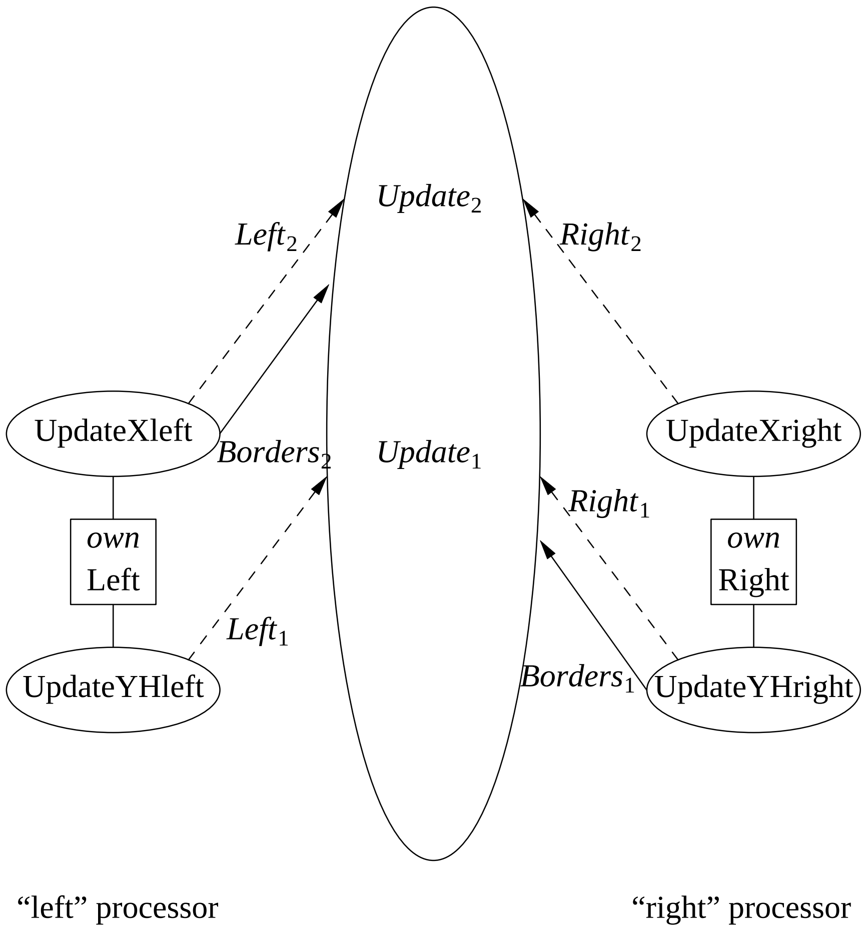


Figure 18 : Communication structure of the tidal model

When transforming the definitions of *Update*$_1$ and *Update*$_2$ to sandwich versions, the normalising variant of the sandwich has to be used, to obtain the correct sequence of both updates. Once the evaluator requires the result of the function *Update*$_1$ (see figure #fig twophase#), reduction continues with the normalisation of the arguments *Left*$_1$, *Right*$_1$ and *BorderOfLeft*$_1$. This normalisation in turn forces the evaluation of *Update*$_2$. So *Update*$_2$ will execute prior to *Update*$_1$. The updating of the x-velocities will run in parallel, yielding the normal forms *Left*$_1$, *Right*$_1$ and *BorderOfLeft*$_1$, directly followed by the parallel updating of the y-velocities and wave heights.

Figure (17) shows the need for the *own* function to avoid redundant data communication. After completion of *UpdateXleft* the resulting matrix *Left*$_1$ is returned and passed unmodified as an argument to *UpdateYHleft*. The updating of matrix *Right*$_2$ follows the same pattern. Both matrices are received as a result to be immediately retransmitted as an argument to the next updating phase. If the functions *UpdateXleft* and *UpdateYHleft* would be evaluated on the same processor, the matrix *Left*$_1$ could be retained in this processor and a remote name could be returned instead. The same applies to the matrix *Right*$_1$ and the functions *UpdateXright* and *UpdateYHright*. The only real data to be returned and retransmitted is the *BorderOfLeft*$_1$, which travels from the “left” processor to the “right” processor. Figure (19) shows the annotation that is necessary to obtain the desired behaviour:

$$\begin{aligned} &UpdateXleft\ Left_2\ BorderOfRight_2 = (own\ Left_1\ ) : RightBorderOf\ \ Left_1 \\ &\qquad \text{WHERE} \\ &\qquad Left_1 = updateXleft\ Left_2\ BorderOfRight_2 \end{aligned}$$

$$UpdateXright\ Right_2 = own\ (updateXright\ Right_2)$$

Figure 19 : Retention of the left matrix

The function *UpdateXleft* returns a remote name for matrix $Left_1$ and real data for the border of $Left_1$. The actual updating takes place in the function *updateXleft* (without capital U). *UpdateXright* just returns a remote name for matrix $Right_1$. Both functions retain the actual matrices in the processors they have been assigned to by the sandwich. Because the remote names $Left_1$ and $Right_1$ are passed as arguments to respectively *UpdateYHleft* and *UpdateYHright*, applications of the latter functions will subsequently be allocated as jobs to the processors where the matrices $Left_1$ and $Right_1$ reside. By retaining the matrices a considerable saving of communication cost is achieved. If the size of the matrix is n then without the own function the amount of data to be communicated would have been $n \times n$, whereas now the information to be transmitted is of the order of n.

The functions of the second updating phase are similar to those of the first phase. Because the main recursion of figure (16) applies $Update_1$ to its own output, one can see that the results of *UpdateYHleft* and *UpdateYHright* are also redirected without any modification into the next iteration of *UpdateXleft* and *UpdateXright*. Figure (20) shows the annotation that is necessary to retain the matrices in their respective processors and to return the actual data of the border of $Right_1$:

$$UpdateYHleft\ Left_2 = own\ (updateYHleft\ Left_2)$$

$$\begin{aligned} &UpdateYHright\ Right_2\ BorderOfLeft_2 = (own\ Right_1\ ) : (LeftBorderOf\ \ Right_1\ ) \\ &\qquad \text{WHERE} \\ &\qquad Right_1 = updateYHright\ Right_2\ BorderOfLeft_2 \end{aligned}$$

Figure 20 : Retention of the right matrix

As before, the update functions (without a capital U) in figure (20) perform the actual updating of the matrices.

The function to force the transmission of the remote matrices at the end of the main recursion is shown in figure (21):

GetRemoteData (*Left* : *Right*) = ‹ *I* *Left* › : ‹ *I* *Right* ›

Figure 21 : Retrieval of both matrices

Both *Left* and *Right* will always be remote names during the iteration of updates, due to the effect of the own function (see figures 19 and 20). The two jobs in figure (21) will therefore be sent to the processors where *Left* and *Right* happen to reside. Upon reception of these jobs the remote names will be deleted and after the evaluation of (*I* *Left*) and (*I* *Right*) the result (*Left* and *Right*) will be returned. No more retention takes place, because the jobs no longer contain the *own* function. Finally the two matrices are paired to represent the state of the tidal model after *n* iterations.

6. Formal description of the transformation schemes

In the previous sections we have presented several examples of application programs with jobs that are annotated by job brackets. Only in the first example (*QuickSort*) the proposed job-lifting and grain size transformations were actually carried out, resulting in a parallel version of the application. In this section we present a formal description of the transformations that is sufficiently general to handle all given example programs.

To describe the job lifting and grain size transformations it is sufficient to define a set of functions operating on restricted syntactic domains.[28] In particular, no knowledge of the expression syntax of SASL is needed. The required domains are listed in figure (22). From the basic domains the abstract syntax shown in the same figure constructs two composed domains: the set of tokens and the set of sequences of tokens. It should be noted that the job brackets are included in the token domain. No higher level syntactic structures need to be recognised in order to describe the transformations.

Syntactic variables and their domains:

I	:	*Identifiers*
L	:	*Literals*
O	:	*Operators*
T	:	*Tokens*
S	:	*Sequences*

Abstract syntax:

S ::= *T* | *S* *T*
T ::= *I* | *L* | *O* | WHERE | = | (|) | → | ; | ,

Figure 22 : Syntactic domains and abstract syntax of a definition with jobs

The two main transformation functions are JL (job-lifting) and GS (grain size). Four additional functions are used: SQ (sequential), L_F (left grain size test), R_F (right grain size test) and A_F (annotation). The latter three functions are application dependent and should be specified for each application separately. That is why their names are provided with a suffix F, which represents the name of the function being transformed. The function A_F decides which of the two sandwich functions to use, either the strict one (*sandwich′*) or the non-strict version (*sandwich*). The left and right grain size tests (R_F and L_F) generate predicates that yield *TRUE* whenever the grain size of the jobs is above an application dependent threshold.

The transformation functions JL, GS and SQ are defined independently of the application by the equations of figure (23). The job lifting function (JL) transforms a given function definition into a version where the two jobs are lifted from a general expression into a single function application. JL also generates a sequential version of the annotated application that will be called when the grain size drops below the threshold. Next the lifted function definition is passed to the grain size transformation (GS), which inserts the grain size tests and the sandwich application. The auxiliary transformation function SQ serves to replace the name of the function being transformed by the unique identifier F_{seq}.

Conventions for variables:			*Conventions for constants:*		
F	:	*Identifiers*	G, P, Q	:	*Identifiers*
T	:	*Tokens*			
a, g	:	*Sequences*			
b, c, d, e, f	:	*Sequences* without occurrences of WHERE or =			

```
JL [[ $F\ a = b ‹ c › d ‹ e › f$  WHERE  $g$ ]] =
        SQ [[ $F$ ]]  [[ $F\ a = b\ (c)\ d\ (e)\ f$  WHERE  $g$ ]]
        GS [[ $F\ a = G\ (c)\ (e)$  WHERE  $G\ P\ Q = b\ P\ d\ Q\ f$
                                             $g$ ]]

JL [[ $F\ a = b ‹ c › d ‹ e › f$ ]] =
      SQ [[ $F$ ]]  [[ $F\ a = b\ (c)\ d\ (e)\ f$ ]]
      GS [[ $F\ a = G\ (c)\ (e)$  WHERE  $G\ P\ Q = b\ P\ d\ Q\ f$ ]]

GS [[ $F\ a = G\ (c)\ (e)$  WHERE  $g$ ]] =
      $F\ a = L_F$ [[ $a$ ]]  [[ $g$ ]]  $\rightarrow$
                  $R_F$ [[ $a$ ]]  [[ $g$ ]]  $\rightarrow$
                        $A_F\ G\ (c)\ (e)$
                  $G\ (c)$  ($SQ$ [[ $F$ ]]  [[ $e$ ]] )
            $R_F$ [[ $a$ ]]  [[ $g$ ]]  $\rightarrow$
                  $G$  ($SQ$ [[ $F$ ]]  [[ $c$ ]] )  $(e)$
            $G$  ($SQ$ [[ $F$ ]]  [[ $c$ ]] )  ($SQ$ [[ $F$ ]]  [[ $e$ ]] )
            WHERE  $g$
```

$$
\begin{aligned}
SQ\,[\![\, F \,]\!]\ [\![\, F\ \ a \,]\!] &= F_{seq}\ \ SQ\,[\![\, F \,]\!]\ [\![\, a \,]\!] \\
SQ\,[\![\, F \,]\!]\ [\![\, T\ \ a \,]\!] &= T\ \ \ \ \ SQ\,[\![\, F \,]\!]\ [\![\, a \,]\!] \\
SQ\,[\![\, F \,]\!]\ [\![\ \ T\ \ ]\!] &= T
\end{aligned}
$$

Figure 23 : Equations job lifting and grain size transformation

The transformation schemes JL and GS can deal with a function that contains more than one equation. The variable F matches the function name in the first equation and the variable a matches all tokens until the equals symbol (=) in the equation with the job brackets (‹ and ›). Similarly the variable g matches all remaining equations.

Figure (24) shows the functions A_F, L_F and R_F to be used for the transformation of the application programs presented in the previous sections. Together with the transformation schemes of figure (23) they generate parallel sandwich versions of the presented application programs. In those cases, where the grain size predicates are identical to the function *TRUE*, the conditional statements generated by the grain size transformation can be simplified.

	L_F	R_F	A_F
QuickSort	*length m > Threshold*	*length n > Threshold*	*sandwich*
FFT	*halfn > Threshold*	*halfn > Threshold*	*sandwich′*
Wang	*TRUE*	*TRUE*	*sandwich′*
Wave $Update_1$	*TRUE*	*TRUE*	*sandwich′*
Wave $Update_2$	*TRUE*	*TRUE*	*sandwich′*
Wave *GetRemoteData*	*TRUE*	*TRUE*	*sandwich′*

Figure 24 : The functions A_F, L_F and R_F for all applications

7. Related work

In our opinion locality is an important concept in computer architecture. For instance the success of virtual memory is largely based on locality in space exhibited by most programs. The current proposal can be classified as a “locality first” design, which makes it different from most contemporary research in the area. Related work will be characterised by the importance attached to the phenomenon of locality in space.

A “divide-and-conquer” combinator was first introduced by Burton and Sleep.[6] The main topics in their paper are network topology and load distribution strategy. A general annotation scheme for the λ-calculus is developed by Burton,[29] which is also applicable to for instance Turner’s combinators. The annotations can be used to control transportation cost of parallel tasks. Although the notion of self contained subexpressions is introduced, the paper does not concern itself with problems associated with practical graph reduction. In recent work, McBurney and Sleep[20] propose a paradigm that models divide-and-conquer behaviour. Their results are based on experiments with transputers but the paradigm is not used in a functional context.

Linear speedups are reported for small programs.

The “RediFlow” architecture[10] provides a global address space, but locality is supposed to be inherent to the function level granularity. Divide-and-conquer applications are mentioned as one possible source of parallelism. The problems associated with a template copying implementation of β-reduction in an implementation of the λ-calculus form one of the major topics of another paper by Keller.[30] The way a closure is implemented brings about some locality.

The “serial” combinator[8, 31] is introduced as an optimal grain of parallelism in the context of fully lazy, parallel graph reduction. The practicality of the approach is demonstrated using a network of processing elements, each with a local store only. The architecture supports a global address space, in which each processing element is responsible for a portion of the store. Locality is supposed to be maintained by the way tasks are diffused to the processing elements to which references exist. In contrast to this approach, the sandwich strategy and job concept may be viewed as a combination of user annotated strictness and user annotated combinators. In addition we propose a “threshold” mechanism to dynamically control the grain size of parallel computations. The *own* function is a user annotated optimisation of data transport.

The “GRIP” proposal[11, 13] avoids the locality issue by using a (high speed) bus as the connection medium between all major system components (processing elements and intelligent storage units). The machine exploits conservative parallel strategies and a “super” combinator[32] model of reduction. In the “FLAGSHIP” machine, both dynamic task relocation and local caches are supposed to increase locality of the fine grained packet rewriting on a local memory architecture.[12]

8. Conclusions

In a parallel graph reduction machine, the optimality of grains of computation depends on properties of the application program and the machine architecture. Based on some commonly observed properties of distributed architectures, a class of application programs has been designated, which if transformed and annotated according to our guidelines will benefit from parallel evaluation on these architectures. In principle our method tries to adapt the locality of the applications to that of the architecture by copying expressions. Duplication of work is avoided by changing the order of the calculations. Suitable grains of parallel evaluation are obtained by grouping certain computations.

Program transformations are necessary to obtain sufficiently large grain computations. With realistic applications these transformations require substantial effort. However because of the referential transparency property of functional programs this effort is less than that incurred in general concurrent programming. It is conceivable that programming tools can be developed to assist the programmer in applying the program transformations, but we have not investigated such possibilities.

The sandwich evaluation strategy bridges the gap between divide-and-conquer algorithms and distributed architectures. The method developed to apply this strategy is independent of the functional programming language used. The proposed evaluation strategy will fit most normal order graph reduction systems.

The practicality of the proposed annotations is demonstrated by transformation of four applications, ranging from the fast Fourier transform to a tidal model, into versions that will run efficiently on parallel machines based on a local memory architecture.

The control over the generation of parallelism and the grain size is exerted by the applications, rather than by the system. Heuristics for grain size control are tailor made to the application program and are therefore a guarantee for best results.

By choosing adequate values for a "threshold" parameter, the maximum number of jobs may be kept within limits acceptable to the concrete architecture. This topic and the two more practical issues related to the optimal value of the threshold (see at the end of section 4.1) will be pursued in part II of this paper.[22]

Acknowledgements

We gratefully acknowledge the support of the Dutch parallel reduction machine project team, in particular the fruitful discussions with Henk Barendregt. Arthur Veen made valuable comments on draft versions of the paper.

References

1. W. E. Kluge, "Cooperating reduction machines," *IEEE transactions on computers* **C-32**(11) pp. 1002-1012 (Nov. 1983).

2. G. A. Magó, "A network of microprocessors to execute reduction languages -- Part I," *J. computer and information sciences* **8**(5) pp. 349-385 (Oct. 1979).

3. G. A. Magó, "A network of microprocessors to execute reduction languages -- Part II," *J. computer and information sciences* **8**(6) pp. 435-471 (Dec. 1979).

4. P. C. Treleaven and R. P. Hopkins, "A recursive computer for VLSI," *Computer architecture news* **10**(3) pp. 229-238 (Apr. 1982).

5. M. Amamiya, "A new parallel graph reduction model and its machine architecture," pp. 1-24 in *Programming of future generation computers*, ed. K. Fychi and M. Nivat, North Holland, Amsterdam, Tokyo, Japan (Oct. 1986).

6. F. W. Burton and M. R. Sleep, "Executing functional programs on a virtual tree of processors," pp. 187-194 in *1st Functional programming languages and computer architecture*, ed. Arvind, ACM, New York, Wentworth-by-the-Sea, Portsmouth, New Hampshire (Oct. 1981).

7. J. Darlington and M. Reeve, "ALICE: {A} multiple-processor reduction machine for the parallel evaluation of applicative languages," pp. 65-76 in *1st Functional programming languages and computer architecture*, ed. Arvind, ACM, New York, Wentworth-by-the-Sea, Portsmouth, New Hampshire (Oct. 1981).

8. P. Hudak and B. F. Goldberg, "Distributed execution of functional programs using serial combinators," *IEEE transactions on computers* **C-34**(10) pp. 881-891 (Oct. 1985).

9. R. M. Keller, G. Lindstrom, and S. Patil, "A loosely-coupled applicative multi-processing system," pp. 613-622 in *48th National computer conf.*, ed. R. E. Merwin and J. T. Zanca, AFIPS, New York (Jun. 1979).

10. R. M. Keller and F. C. H. Lin, "Simulated performance of a reduction based multiprocessor," *IEEE computer* **17**(7) pp. 70-82 (Jul. 1984).

11. S. L. Peyton Jones, "Using Futurebus in a fifth-generation computer," *Microprocessors and microsystems* **10**(2) pp. 69-76 (Mar. 1986).

12. P. Watson and I. Watson, "Evaluation of functional programs on the Flagship machine," pp. 80-97 in *3rd Functional programming languages and computer architecture, LNCS 274*, ed. G. Kahn, Springer-Verlag, Berlin, Portland, Oregon (Sep. 1987).

13. S. L. Peyton Jones, "Directions in functional programming research," pp. 220-249 in *SERC conf. on distributed computing systems programme*, ed. D. A. Duce, Peter Peregrinus, Brighton, England (Sep. 1984).

14. T. Johnsson, "Efficient compilation of lazy evaluation," *ACM SIGPLAN notices* **19**(6) pp. 58-69 (Jun. 1984).

15. T. H. Brus, M. C. J. D. van Eekelen, M. O. van Leer, and M. J. Plasmeijer, "CLEAN: {A} language for functional graph rewriting," pp. 364-384 in *3rd Functional programming languages and computer architecture, LNCS 274*, ed. G. Kahn, Springer-Verlag, Berlin, Portland, Oregon (Sep. 1987).

16. J. Fairbairn and S. C. Wray, "Tim: {A} simple lazy abstract machine to execute supercombinators," pp. 34-45 in *3rd Functional programming languages and computer architecture, LNCS 274*, ed. G. Kahn, Springer-Verlag, Berlin, Portland, Oregon (Sep. 1987).

17. D. A. Turner, "Functional programs as executable specifications," pp. 29-54 in *Mathematical logic and programming languages*, ed. C. A. R. Hoare and J. C. Shepherdson, Prentice Hall, London, England (Feb. 1984).

18. D. A. Turner, "A new implementation technique for applicative languages," *Software−practice and experience* **9**(1) pp. 31-49 (Jan. 1979).

19. C. A. Ruggiero and J. Sargeant, "Control of parallelism in the Manchester data flow machine," pp. 1-15 in *3rd Functional programming languages and computer architecture, LNCS 274*, ed. G. Kahn, Springer-Verlag, Berlin, Portland, Oregon (Sep. 1987).

20. D. L. McBurney and M. R. Sleep, “Transputer based experiments with the ZAPP architecture,” pp. 242-259 in *1st Parallel architectures and languages Europe (PARLE), LNCS 258/259*, ed. J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, Springer-Verlag, Berlin, Eindhoven, The Netherlands (Jun. 1987).

21. D. A. Turner, “SASL language manual,” Technical report, Computing Laboratory, Univ. of Kent at Canterbury (Aug. 1979).

22. P. H. Hartel and W. G. Vree, “Parallel graph reduction for divide-and-conquer applications -- Part II: program performance,” PRM project internal report, Dept. of Comp. Sys, Univ. of Amsterdam (Dec. 1988).

23. G. D. Bergland, “Fast Fourier transform hardware implementations -- an overview,” *IEEE transactions on audio and electro acoustics* **AU-17** pp. 104-108 (Jun. 1969).

24. H. H. Wang, “A parallel method for tri-diagonal equations,” *ACM transactions on mathematical software* **7**(2) pp. 170-183 (Jun. 1981).

25. P. H. Michielse and H. A. van der Vorst, “Data transport in Wang’s partition method,” Internal report, Dept. of Comp. Sci, Technical Univ. Delft (1986).

26. W. G. Vree, “The grain size of parallel computations in a functional program,” pp. 363-370 in *Parallel processing and Applications*, ed. E. Chiricozzi and A. d’Amico, Elsevier Science Publishers, L’Aquila, Italy (Sep. 1987).

27. W. G. Vree, “Parallel graph reduction for communicating sequential processes,” PRM project internal report, Dept. of Comp. Sys, Univ. of Amsterdam (Aug. 1988).

28. R. D. Tennent, “The denotational semantics of programming languages,” *CACM* **19**(8) pp. 437-453 (Aug. 1976).

29. F. W. Burton, “Annotations to control parallelism and reduction order in the distributed evaluation of functional programs,” *ACM transactions on programming languages and systems* **6**(2) pp. 159-174 (Apr. 1984).

30. R. M. Keller, “Distributed graph reduction from first principles,” pp. 1-14 in *Implementation of functional languages*, ed. L. Augustsson, R. J. M. Hughes, T. Johnsson, and K. Karlsson, Programming Methodology group report 17, Dept. of Comp. Sci, Chalmers Univ. of Technology, Goteborg, Sweden, Aspenas, Sweden (Feb. 1985).

31. P. Hudak and B. F. Goldberg, “Serial combinators: ‘‘Optimal’’ grains of parallelism,” pp. 382-399 in *2nd Functional programming languages and computer architecture, LNCS 201*, ed. J.-P. Jouannaud, Springer-Verlag, Berlin, Nancy, France (Sep. 1985).

32. R. J. M. Hughes, “Super combinators -- A new implementation method for applicative languages,” pp. 1-10 in *Lisp and functional programming*, ACM, New York, Pittsburg, Pennsylvania (Aug. 1982).

CHAPTER VI

THE GRAIN SIZE OF PARALLEL COMPUTATIONS IN A FUNCTIONAL PROGRAM[1]

[1] in Conf. on parallel processing and applications, l'Aquila, Italy, September 1987, Elsevier Science Publishing, pp 363-370

The Grain Size of Parallel Computations in a Functional Program.

Willem G. Vree
Computer Science Department of the Dutch Water Board Authority [1]
Nijverheidsstraat 1
Post Box 5809, 2280HV Rijswijk ZH, The Netherlands

Abstract

A parallel functional program to predict the tides of the North Sea is developed in three stages. The first version of the program is obtained by a direct translation of the mathematical model. Two successive transformations appear to be necessary to remove inefficiencies and to enlarge the grain size of parallel computations. The enlargement of parallel grains is needed to make the program well suited for coarse grain architectures. The method to enlarge the grain size, called data grouping, can be elegantly expressed in a functional language and has a wide range of applications.

1. Introduction

Functional languages are often considered to be well suited for parallel machine architectures [VEG84]. At any given stage during its evaluation a functional program may contain several function applications that can be rewritten (reducible expressions, or shorter: redexes). The Church-Rosser property of functional languages offers the theoretical possibility to evaluate these redexes in parallel. However, there are two major problems that limit the exploitability of the available parallelism:

1• Not all function applications are needed to compute the final result. So indiscriminate (parallel) evaluation of all redexes may lead to a waste of computing resources.

2• Most function applications will not contain a sufficient amount of computation to justify the overhead that is incurred by data transmission and process synchronization in parallel

[1] Present address of the author is University of Amsterdam, FVI, Post Box 41882, 1009DB Amsterdam, The Netherlands

machine architectures. The grain size of a function application will be defined as the amount of computation needed to evaluate this application.

The first problem has given rise to the development of strictness analysis of functional programs. The strictness of a function in its arguments is an undecidable property and can to some extent be determined in flat domains by the technique of abstract interpretation [MYC81]. The second problem of grain size in functional programs, has received some attention [HUD85,HUG84,KEL84]. Also the grain size of a function application is an undecidable property and has to be compared to the inaccurate notion of communication cost and process synchronization overhead.

2. Grain size enlargement

This paper concentrates on the grain size problem in relation to a specific class of MIMD-architectures. In these architectures the cost to transport an elementary data item is large compared to the cost to execute an elementary machine instruction (e.g. a reduction step). Many MIMD-reduction machines that are currently under development belong to this class [BAR87]. It will be demonstrated, with the aid of an example program of moderate size, that it is sometimes necessary to perform complicated program transformations in order to obtain a useful grain size for these MIMD architectures. This claim is made by considering the source text of the program and determining the largest possible grain of parallel computation. Because this grain size appears to be still too small for the considered class of architectures, a program transformation has to enlarge the grain size.

The grain size that would be obtained by the translation of a program into super-combinators [HUG84] or serial-combinators [HUD85] is bound to be smaller (or equal) than the largest possible grain size at source text level.

The example program has been developed as a test case for the current design of an experimental MIMD-reduction machine for the Dutch Parallel Reduction Machine project. The program is a simplified version of a production program used by the Dutch Water Board Authority to predict the tides of the North Sea.[HEE85,86] Although simplified it is representative for the original program in the sense that speed-up by parallel execution, will also hold for the original program. It contains sufficient details (like influence of wind, coriolis force, bottom friction etc.) to make a reasonably accurate model of the tides.

Apart from the contribution to the discussion on the grain size problem the paper also adds a (parallel) program to the small number of existing functional programs that could serve to test the validity of parallel reduction architectures

3. The physical problem

This section describes the basic knowledge of the application domain that is necessary to understand the example program. The water movement in an estuary is approximated with a mathematical model in two spatial dimensions. This means that the velocity of the water is assumed not to vary in the vertical dimension and so only very large waves, like the tidal waves will be modeled appropriately. Figure 1 shows the equations that constitute the approximated model.

$$\frac{\partial u}{\partial t} + \mathrm{g}\,\frac{\partial h}{\partial x} - \mathrm{f\,v} + \lambda\,\frac{u}{D} - \gamma\,\frac{V^2 \cos Y}{D} = 0$$

$$\frac{\partial v}{\partial t} + \mathrm{g}\,\frac{\partial h}{\partial y} + \mathrm{f\,u} + \lambda\,\frac{v}{D} - \gamma\,\frac{V^2 \sin Y}{D} = 0$$

$$\frac{\partial h}{\partial t} + \frac{\partial (Du)}{\partial x} + \frac{\partial (Dv)}{\partial y} = 0$$

where:

h = small variations in the water height

u = water velocity in the x-direction

v = water velocity in the y-direction

g = acceleration of gravity (9.8 ms^{-2})

f = coriolis parameter (1.25 10-4 s^{-1})

λ = bottom friction coefficient (±2.4 10^{-3} ms^{-1})

γ = wind transfer coefficient (±3.2 10^{-6})

D = water depth as function of x and y

ψ =wind direction

V = wind velocity

FIGURE 1: The linearized shallow water equations.

The first two equations state that the water velocity is proportional to the gradient of the water height and the effects of the earth rotation (f), bottom friction (λ) and wind (γ) are included. The third equation expresses the conservation of mass. It states that a disappearing quantity of water will result in a decreasing water height.

The numerical approximation of the proposed equations that we have used in the example program is presented in figure 3. (for a derivation see [HOU68]) The variables from the equations of figure 1 (u, v and h) have been approximated by their values on a spatial grid (subscripts i, j respectively in the x, y direction) at discrete points in time (superscript k). Figure 2 shows that each of the variables u,v,h and the depth D has its own grid that is slightly shifted with respect to the others. Such a space-staggered grid allows for an easy boundary treatment.

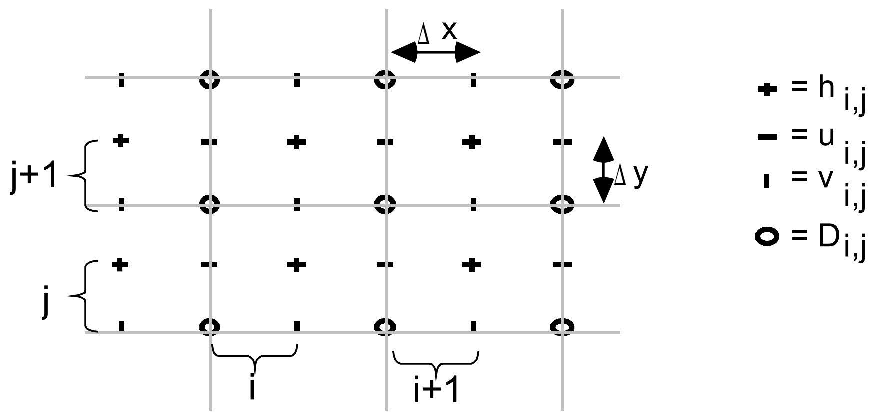


FIGURE 2: The space-staggered grid

$$u^{k+1}_{i,j} = u^{k}_{i,j} - g\frac{Dt}{2Dx}(h^{k}_{i,j} - h^{k}_{i-1,j}) + f\,\frac{Dt}{4}(v^{k}_{i-1,j} + v^{k}_{i-1,j+1} + v^{k}_{i,j} + v^{k}_{i,j+1}) - 2\Delta t\left\{\frac{lu^{k}_{i,j} - V^{2}\cos Y}{D_{i,j} + D_{i,j+1}}\right\}$$

$$v^{k+1}_{i,j} = v^{k}_{i,j} - g\frac{Dt}{2Dy}(h^{k}_{i,j} - h^{k}_{i,j-1}) + f\,\frac{Dt}{4}(u^{k+1}_{i,j-1} + u^{k+1}_{i+1,j-1} + u^{k+1}_{i,j} + u^{k+1}_{i+1,j}) - 2\Delta t\left\{\frac{lv^{k}_{i,j} - V^{2}\sin Y}{D_{i,j} + D_{i+1,j}}\right\}$$

$$h^{k+1}_{i,j} = h^{k}_{i,j} - \frac{Dt}{4Dx}\left\{\left(D_{i+1,j} + D_{i+1,j+1}\right)u^{k+1}_{i+1,j} - \left(D_{i,j} + D_{i,j+1}\right)u^{k+1}_{i,j}\right\}$$
$$- \frac{Dt}{4Dy}\left\{\left(D_{i,j+1} + D_{i+1,j+1}\right)v^{k+1}_{i,j+1} - \left(D_{i,j} + D_{i+1,j}\right)v^{k+1}_{i,j}\right\}$$

FIGURE 3: the finite difference scheme

It turns out [HEE85,86] that the system of figure 3 is stable under the following condition :

$$\Delta t < 2\ \Delta x\ \Delta y\ \left(gD\left(Dx^{2} + Dy^{2}\right)\right)^{-1/2}$$

Based on the finite difference scheme the example program will now be developed in three stages. The first version of the program is the direct translation of the finite difference scheme into SASL and the two subsequent versions are produced by successive transformation of the previous versions. After each stage the amount of parallelism and the grain size will be discussed.

4. The first program

Because the finite difference scheme is similar to a set of recursion equations, they almost constitute a functional program. For the simple case of a rectangular grid, the corresponding SASL program is presented in figures 4 and 5. To obtain this program all sub- and superscripted variables (u, v, h and D) in figure 3 are replaced by a function of their sub- and superscripts, in the following way:

$$u^{k+1}_{i,j} \Rightarrow \mathrm{u\ i\ j\ k}\quad v^{k+1}_{i,j} \Rightarrow \mathrm{v\ i\ j\ k}\quad h^{k+1}_{i,j} \Rightarrow \mathrm{h\ i\ j\ k}\quad D_{i,j} \Rightarrow \mathrm{D\ i\ j}$$

As an example figure 4 shows the SASL[1] function corresponding to the variable u that is obtained in this manner. The only additions to the transformed equation are the two initial "if"-statements. The first one terminates the recursion on the discrete time variable k. The second "if" implements the boundary condition of a rectangular grid. (note that only i needs to be tested, due to the space staggered grid of figure 2 [HEE85,86]).

```
u i j k   =     k = 0 -> u0  i j
                i = 0 | i = imax -> 0
                u i j (k-1) - heightgradient + coriolis - friction
                WHERE
                        heightgradient = g * Δt / (2 * Δx) * ( h i j (k-1) - h (i-1) j (k-1))
                        coriolis = f * Δt / 4 * (v (i-1) j (k-1) + v (i-1)(j+1)(k-1) + v i j (k-1) + v i (j+1)(k-1))
                        friction = 2 * Δt * (λ * u i j (k-1) - wind) / (D i j+ D i (j+1))
                        wind = V * V * cos( Ψ )
```

FIGURE 4: SASL function for the x-velocity u.

In the same way, the functions corresponding to the variables v and h, are constructed from the equations of figure 3 (the second "if" for v should now test j, and no boundary test is needed for h). The SASL program is completed by including the initial values of u,v and h and all constants (as an example see figure 5).

```
u0  i j = 0                  || initial values of u e.g. all 0
v0  i j = 0                  || initial values of v e.g. all 0
h0  i j = 3 * i / (imax-1)   || initial values of h, e.g. a water slope
imax = 100                   || rectangular grid with 100 * 100 gridpoints
jmax = 100
Δx = Δy =10000               ||  = 10 km, so one side of the square estuary  is Δx * imax = 1000 km
Δt  = 800                    || one time step is 800 seconds; the stability condition is satisfied
D i j = 30                   ||  Depth function, e.g. constant depth of 30 meters.
|| the other constants ( for their values see fig 1)  have to be included here
```

FIGURE 5: the first program.

Although the translation of the program closely resembles the original equations, it suffers from a serious drawback, namely the recomputation of function applications. From the definition of h and u it follows that: (see figure 3 and 4)

$$\left.\begin{array}{l} h\ i\ j\ k \text{ needs the value of } h\ i\ j\ (k\text{-}1),\ u\ (i\text{+}1)\ j\ k,\ u\ i\ j\ k \\ u\ i\ j\ k \text{ needs the value of } h\ i\ j\ (k\text{-}1),\ h\ (i\text{-}1)\ j\ (k\text{-}1) \end{array}\right] \Rightarrow$$

$$h\ i\ j\ k \text{ needs the value of } h\ i\ j\ (k\text{-}1), h\ (i\text{-}1)\ j\ (k\text{-}1),\ h\ (i\text{+}1)\ j\ (k\text{-}1) \qquad 1)$$

[1] We will use both Greek and Roman characters in identifiers and in section 5 we will extend SASL with a notation for array subscription.

and rewriting *h (i-1) j (k-1)* with 1) →

h i j k needs the value of *h i j (k-1), h i j (k-2)*

So in the mutual recursive scheme hides a recursion of the Fibonacci type and in fact there is such a recursion for each variable, giving rise to exponential recalculation of function applications. One solution to this problem could be the use of an applicative cache [KEL81], but a cache introduces two other problems. On the one hand, implementing a global cache in the considered class of MIMD-architectures, causes a lot of communication overhead. On the other hand it complicates reasoning about grain size of computations.

Another solution could be the use of "memo-functions" [TUR81], but in this application they appear to require an unreasonable storage capacity because the values of u,v and h will be "remembered" in all grid points at all time steps. Instead we propose a transformation of the program into a finite state machine, which might also be considered as a kind of "memo-function" but with a restricted short term memory (i.e. the state).

5. Transformation to the second version

The second version of the program is obtained from the first one by ordering the calculations in such a manner that recomputations will be avoided. The set of recursive difference equations is transformed into a finite state machine, where the state consists of a matrix containing the values of the variables u,v and h in each grid point (figure 6). The program then repeatedly distributes the same calculation over all matrix elements (like the map function) and proceeds until the desired final state has been reached. The successive creation of new matrices (by the repeated distribution) does not compare unfavourably to an imperative program doing destructive updates, because of the following two reasons:

1) For each iteration the algorithm requires all matrix elements to be recalculated, so there is no needless structure copying.

2) Once the new matrix has been calculated there are no references left to the old one and memory space can be recovered.

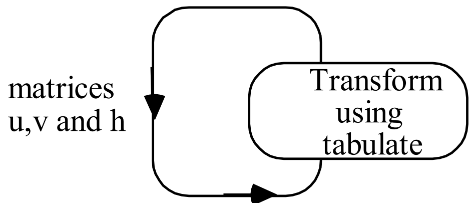


FIGURE 6: A finite state machine

To deal efficiently with matrices we introduce an array data type, where subscription can be performed in constant time. An array will be characterized by a descriptor, containing the upper and lower limit of the index in each dimension [1]:

$$\text{descriptor} = (l_1,u_1),(l_2,u_2)$$

Subscription of the array variable u is denoted by u[i,j] and results in the value $u_{i,j}$ in constant time. "Tabulate f d" is a special function that tabulates a binary function f over all index pairs specified by descriptor d. It constructs a 2-dimensional such that:

$$(\text{tabulate f d})\ [\ i,j\ ] = \text{f i j}\ ,\quad l_1 \leq i \leq u_1\ ,\quad l_2 \leq j \leq u_2$$

"Tabulate" is the source of parallelism in the finite state machine of figure 6. The function applications (f i j) form the largest grain of parallel computation and should be distributed by "tabulate" over the available processors in the MIMD architecture.

Figure 7 shows the second program, where the functions "repeat" and "transform" define the finite state machine. The functions fu,fv and fh can be derived from the previous program. For instance, the body of (fu u v h i j) in figure 7 can be obtained from the body of (u i j k) in figure 4 by replacing each occurrence of "i j (k-1)" by "[i,j]". The definitions of fv and fh are derived in the same way.

[1] The array data type is shown for the case of two dimensions because the example program only needs two dimensional arrays.

```
solution  n =  repeat n (uinit,vinit,hinit)   ||  u,v,h after n time steps

repeat 0  ( u, v, h )  = ( u, v, h )
repeat n  ( u, v, h )  = repeat (n-1) (transform u v h)

dscru = (0,imax     ) , (0,jmax -1)            || array descriptor of u
dscrv = (0,imax -1) , (0,jmax     )            || array descriptor of v
dscrh = (0,imax -1) , (0,jmax -1)              || array descriptor of h

transform  u v h                  || calculates the matrices at the next time step
                                          || (u1,v1,h1) from the current ones (u,v,h)
          =         ( u1, v1, h1 )
                    WHERE
                              u1 = tabulate (fu  u   v   h)  dscru
                              v1 = tabulate (fv  u1 v   h)  dscrv
                              h1 = tabulate (fh  u1 v1 h)  dscrh

fu  u v h i j
          =         i = 0 | i = imax -> 0
                    u[ i, j ] - heightgradient + coriolis - friction
                    WHERE
                              heightgradient = g * Δt / (2 * Δx) * ( h[ i, j ] - h[ i-1, j ] )
                              coriolis = f * Δt / 4 * (v[ i-1, j ] + v[ i-1, j+1] + v[ i, j ]+ v[ i, j+1 ] )
                              friction = 2 * Δt * (λ * u[ i, j ] - wind) / (D i j + D i (j+1) )
                              wind = V * V * cos( Ψ )

fv  u v h i j       || constructed like (fu u v h i j) from the body of (v i j k)

fh u v h i j        || constructed like (fu u v h i j) from the body of (h i j k)

uinit     = tabulate u0  dscru                || initial values of u,v and h
vinit     = tabulate v0  dscrv
hinit     = tabulate h0  dscrh

||the rest of the definitions are identical to those needed in figure 5
```

FIGURE 7: The second program

Although the program of figure 7 does not recompute function applications it is still far from being well suited for large-grain parallel distribution. This is because the functions fu,fv and fh which are distributed by tabulate do not have recursion. Worse yet they contain a lot of small array references (that might cause data communication). There is no way to arrange for an efficient implementation of tabulate under these circumstances. The most efficient way is probably to cut the matrices into regular pieces, distribute these parts and to require a remote processor to perform the tabulate on that piece it happens to receive. However, when the array references do not exhibit locality, each processor will have to make as many global references as it received array elements and thus communication cost grows as fast as the amount of computation. (i.e. proportional to the number of array elements). If the array references do have locality, then an implementation of tabulate cannot know how to divide the matrices without corrupting locality.

6. Transformation to the third version

The next transformation constructs a course grain parallel program by using the presence of locality in the grid point calculations of the previous program. Because of this locality it is possible to split the original finite state machine (of figure 6) into several communicating finite state machines, without a significant increase of communication cost. This is accomplished by dividing the original matrices (u,v and h) into several subparts and to associate with each submatrix a function that is almost identical to the program of figure 7. To save space and to gain clarity the transformation is only elaborated for the simplest case of two submatrices (see figure 8), but extension to more submatrices is straight forward. The number of parallel processes (submatrices) is limited by a communication-processing trade-off. The amount of computation per process is proportional to the number of grid points in a matrix partition and grows with the square of the size of the partition. The amount of communication, however, is proportional to the number of border grid points and only grows linearly with the size of the partition. Given a sufficiently large problem, this property theoretically allows the program to be adjusted to any communication speed and any amount of parallelism.

It is important to notice that the functions fu,fv and fh of the second program can be used without modification in the third version i.e. the transformation merely adds a layer to the program. This layer describes a set of communicating processes, each of which comprises the unmodified functions of the second program. This elegant structure might well be attributed to the hierarchical structure of functional programs.

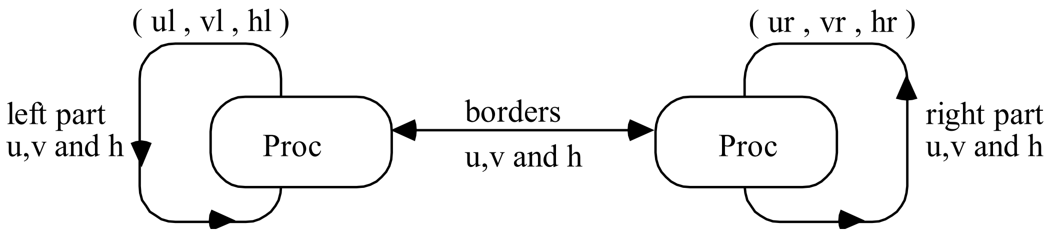


FIGURE 8: A partition in two processes

Both processes in figure 8 are course grain, as they are continuously updating half of the original matrices. They can also be conveniently distributed to different processors as only their borders have to be transmitted after each time step. The partition of the matrices is based on the precise lay-out of the space staggered grid. Without going onto details, one can compare the descriptors of the left and right matrices in figure 10 with the illustration of the partitioned grid in figure 9. The column ul_k and the columns vr_0 ,hr_0 in figure 9 constitute the borders that have to be communicated between the left-and right process.

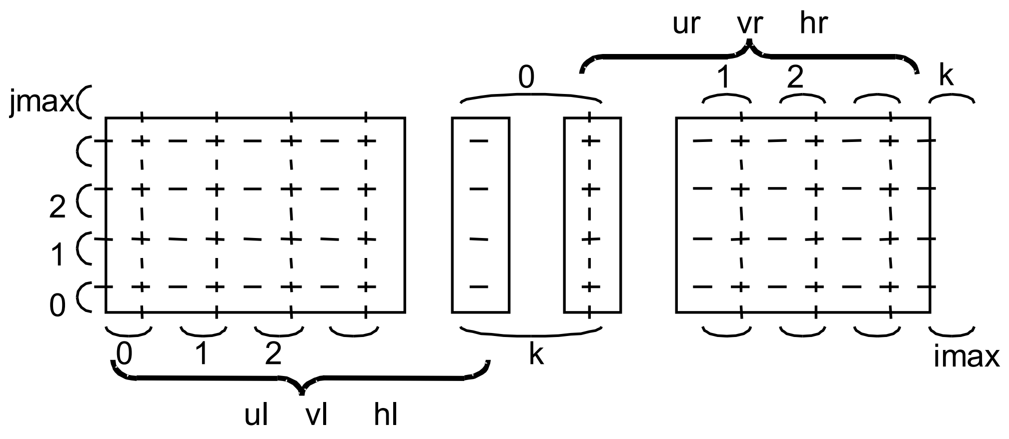


FIGURE 9: The grid partition

```
k        = imax / 2                         || imax assumed to be even
dul      = (0      ,k      ) , (0,jmax -1)  || descriptor of u-left
dvl      = (0      ,k - 1) , (0,jmax     )  || descriptor of v-left
dhl      = (0      ,k - 1) , (0,jmax - 1)   || descriptor of h-left
dur      = (k + 1,imax) , (0,jmax - 1)      || descriptor of u-right
dvr      = (k,imax - 1) , (0,jmax     )     || descriptor of v-right
dhr      = (k,imax - 1) , (0,jmax - 1)      || descriptor of h-right
ul0      = tabulate u0 dul                  || initial u-left matrix
ur0      = tabulate u0 dur                  || initial u-right matrix
|| similar definitions for vl0,vr0,hl0 and hr0 using their corresponding descriptors
|| the rest of the definitions are identical to those of figure 5
```

FIGURE 10 : the descriptors and initial values of the partitioned matrices.

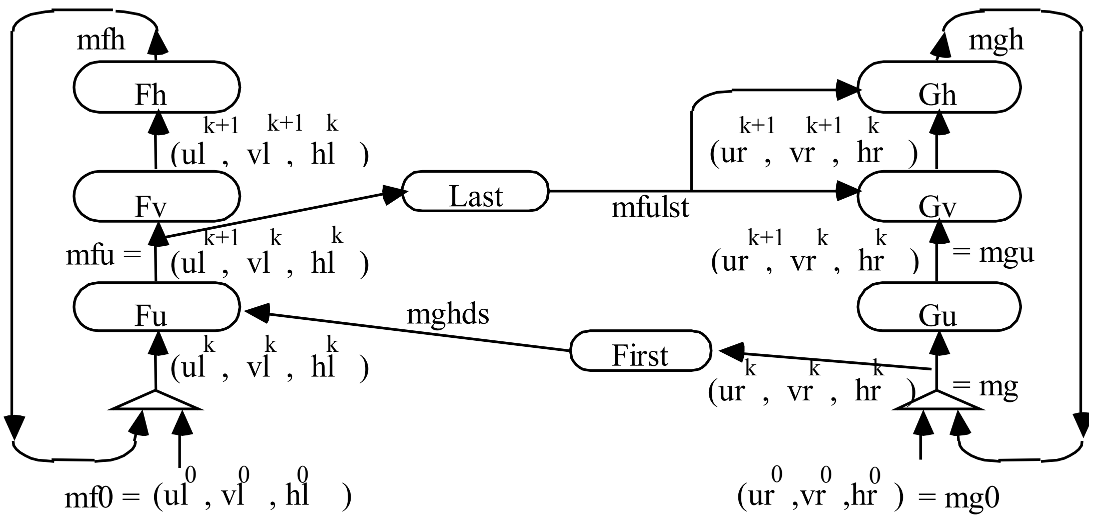

mfh
Fh
$(ul^{k+1}, vl^{k+1}, hl^{k})$
Fv
Last
mfulst
mfu = $(ul^{k+1}, vl^{k}, hl^{k})$
Fu
mghds
First
(ul^{k}, vl^{k}, hl^{k})
mf0 = (ul^{0}, vl^{0}, hl^{0})
mgh
Gh
$(ur^{k+1}, vr^{k+1}, hr^{k})$
Gv
$(ur^{k+1}, vr^{k}, hr^{k})$ = mgu
Gu
(ur^{k}, vr^{k}, hr^{k}) = mg
(ur^{0}, vr^{0}, hr^{0}) = mg0

FIGURE 11: the structure of the third program (refinement of figure 8).

The program structure is illustrated in figure 11. For the sake of clarity the three matrices u, v and h have been grouped together (matrix triplet) and are passed as such from function to function, although each function only updates one of the three matrices. The left and right process each consist of three functions, interconnected by infinite lists (streams) of matrix-triplets. The two triangles at the bottom of figure 11 represent the initial pairing function that prefixes the infinite lists (mfh and mgh) with the initial matrices. The functions Fu,Fv and Fh process the left matrix-triplet stream. They contain respectively the unmodified functions fu,fv and fh of the second program (figure 7). Gu,Gv and Gh process the right triplet stream and also contain fu,fv and fh.

Communication between the two processes is performed by the function 'First', which continuously transmits the first columns of 'vr' and 'hr', and the function 'Last', which continuously transmits the last column of 'ul'. The SASL program corresponding to the illustration of figure 11 is shown in figure 12:

```
solution  k = mfh   k , mgh   k                  || the estuary after  k  time steps

mfh      = Fh ( Fv  mfu)                         || the left process
mfu      = Fu ( mf0 : mfh )  mghds
mfulst   = Last  mfu
mgh      = Gh  mfulst  ( Gv mfulst mgu )         || the right process
mgu      = Gu  mg
mg       = mg0 : mgh
mghds    = First  mg
mf0      = ( ul0, vl0, hl0 )                     || the initial triplets
mg0      = ( ur0, vr0, hr0 )

Fu (( u,v,h ) : Restuvh )  (( vc,hc ) : RestvhCol)
         =        ( u1,v,h) :  Fu  Restuvh  RestvhCol
                  WHERE         u1 = tabulate ( fu u (appendcol v vc) (appendcol h hc))  dul

Fv  (( u,v,h ) : Restuvh )
         =        ( u,v1,h ) : Fv  Restuvh
                  WHERE         v1 = tabulate ( fv u v h )  dvl

Fh  (( u,v,h ) : Restuvh )
         =        ( u,v,h1 ) : Fh  Restuvh
                  WHERE         h1  = tabulate ( fh u v h )  dhl

Gu  (( u,v,h ) : Restuvh )
         =        ( u1,v,h ) :  Gu  Restuvh
                  WHERE         u1 = tabulate ( fu u v h )  dur

Gv  (( u,v,h ) : Restuvh ) ( uc : RestuCol )
         =        ( u,v1,h ) : Gv  Restuvh  RestuCol
                  WHERE         v1 = tabulate ( fv (prependcol u uc) v h )  dvr

Gh  (( u,v,h ) : Restuvh) ( uc : RestuCol )
         =        ( u,v,h1 ) : Gh  Restuvh  RestuCol
                  WHERE         h1 = tabulate ( fh (prependcol u uc) v h )  dhr

First  (( u, v, h ) : Restuvh ) = (firstcol v, firstcol h) : First Restuvh
Last (( u,v,h ) : Restuvh ) = lastcol u : Last Restuvh
firstcol  matrix =              || returns the first column of matrix
lastcol  matrix =               || returns the last column of matrix
appendcol  matrix  col =        || appends column after the last column of matrix
prependcol  matrix  col =       || prepends column before the first column of matrix
|| the definitions of figure 7 and 10 should be included here
```

FIGURE 12: the third program .

To run the program on a two processor system, the function applications defining the streams "mfh, mfu and mfulst" should be evaluated on one processor and those defining "mgh, mgu, mg and mghds" on the other processor. In a parallel implementation of SASL, these streams can be marked, to communicate the programmers intentions to the compiler. The proposed annotation of streams serves two purposes: it indicates the coarse grains of parallelism and it prescribes the static distribution of these grains.

It is interesting to know how much the second program is slowed down by the addition of the distribution layer. Counting reduction steps of both programs on the same input showed an overhead of 2%. A simulation of a 10^4 km^2 estuary during one hour of physical time took respectively 201597 and 204719 combinator reduction steps for program 2 and 3.

7. Data grouping

The essential part of the technique that is used in section 6 to enlarge the grain size of parallel computations, can be emphasized by giving a one dimensional example without recursion:

ParTabulate F (1..10) ⇒ SeqTabulate F (1..5) *in parallel with* SeqTabulate F (6..10)

The function ParTabulate is supposed to distribute all the applications (F i), for i=1..10 over the available processors, yielding ten parallel grains with a size of one application of F. In the transformed program, however, SeqTabulate will perform five applications of F sequentially, resulting in two parallel grains of five applications of F. We would like to call this method "data grouping" because many fine grained applications are grouped into one larger grain.

8. Conclusion

A program of moderate size and complexity (a model of the tides in the North Sea) has been developed in SASL, containing a flexible grain size that can be adjusted to fit a large class of MIMD-architectures. It is demonstrated that in a functional language the program can be developed in a systematical way. Two successive transformations are applied to a program that is obtained by a direct translation of the mathematical model into SASL. The first transformation removes inefficiencies due to exponential recalculation of function applications. The second transformation, called "data grouping", enlarges the grain size of parallel computations.

In a functional language, the latter transformation can be added to the sequential program as a separate layer. This distribution layer can be elegantly expressed as a set of concurrent processes, communicating via streams. Annotation of the obtained coarse grain parallelism (required for a practical parallel implementation) by marking of the appropriate streams, also indicates the static distribution of these grains.

The data grouping transformation has a wide range of applications, in particular those based on regular grid calculations (e.g. immage processing).

9. Acknowledgements

I wish to express my gratitude to Arnold Heemink for his clear explanations of the shallow water equations and their numerical approximation. Due to fruitful discussions with him I was able to write the example program. I am also much indebted to the Dutch Water Board Authority who kindly permitted me to work on the Dutch Parallel Reduction Machine Project. Finally I would like to thank Pieter Hartel for his valuable comments during the preparation of this article.

10. References

[BAR87] H.P.Barendregt et al.,"The Dutch parallel reduction machine project", submitted to Frontiers in Computing, Amsterdam, Dec 1987

[MYC81] A.Mycroft,"Abstract interpretation and optimising transformations for applicative programs",PhD. thesis, Univ. of Edinburgh, 1981

[HEE85] A.W.Heemink,"Application of Kalman filtering to tidal flow prediction",Proc. IFAC/IFORS, Lissabon 1985

[HEE86] A.W.Heemink,"Storm surge prediction using Kalman filtering",PhD. thesis,Twente Technical University,Holland, sept 1986

[HOU68] P.J.van der Houwen,"Finite difference methods for solving partial differential equations",Mat. Cen. Tracts, 20, Mat. Cen.,Amsterdam 1968

[HUD85] P.Hudak,"Distributed execution of functional programs using serial combinators",IEEE Trans. on Computers, October 1985

[HUG84] J.Huges,"Graph reduction with super-combinators",rep PRG-28,Oxford University, June 1982

[KEL84] R.M.Keller,F.C.H.Lin,"Simulated performance of a reduction based multiprocessor",IEEE comput.,vol 17,pp 70-82,July 1984

[KEL81] R.M.Keller,M.R.Sleep,"Applicative Caching",Proc. ACM conf. functional programming lang. & comp. arch.,1981, pp131-140

[TUR81] D.A.Turner,"The semantic elegance of applicative languages",Proc. of the ACM conference on Functional programming languages and computer architecture, Porthmouth, Oct 1981

[VEG84] S.R.Vegdahl,"A survey of proposed architectures for executing functional languages",IEEE trans. on computers, December 1984

[VRE87] W.G.Vree,"A parallel hydrolical simulation program in sasl", internal report D-13, university of Amsterdam, 1987

CHAPTER VII

PARALLEL GRAPH REDUCTION FOR SYNCHRONOUS PROCESS NETWORKS[1]

[1] PRM project internal report D-26, dept.of Comp. Sys., Univ. of Amsterdam, Februari 1989

Parallel Graph Reduction for Synchronous Process Networks

Willem G. Vree

Computer systems department, University of Amsterdam

Kruislaan 409, 1098 SJ Amsterdam

email: wimv@uva.uucp

abstract

Process networks can be elegantly expressed in functional languages by tail-recursive functions interconnected by lazy streams. In a graph reduction system these kind of application programs give rise to cyclic graphs. It is shown that a network of synchronous processes with possibly cyclic interconnections can be transformed into a single acyclic synchronous process by eliminating all streams. A formal definition of this transformation that we call communication lifting is presented. As an example two application programs (a tidal model for the North Sea and a simulation of digital hardware) are transformed and mapped onto a coarse grain parallel reduction model that only supports strict argument parallelism. Such a mapping is not possible for the original application programs, because there is no way to express the required "pipe-line" parallelism of streams in a reduction model only based on strict operator parallelism.

1 Introduction

Functional languages with normal order semantics can be implemented efficiently on sequential architectures by graph reduction [PEY87a]. The program is represented by a graph data-structure in which data and computations may be shared. The basic mechanism of computation is the rewriting of parts of the graph according to certain *graph rewrite rules*. A sub-graph that can be rewritten according to such a rule is called a *redex* (red̲ucible ex̲pression). The process of rewriting is repeated until a certain halting criterion is met. For instance, reduction may be stopped when the root node of the graph is no part of any redex. The graph is then said to be on *root normal form* (also called *head normal form* in the world of term rewriting). In section 3 we introduce a notation for graphs and graph rewrite rules adopted from CLEAN [BRU87, BAR87a,b], which is an intermediate language specially designed to study both theoretical and practical properties of graph reduction. It will be used to denote rewrite rules and to describe the reduction behaviour of application programs based on concurrent processes.

In practical graph rewrite systems reducible expressions are always *disjoint* (because the rules are weakly regular [BAR87a]). This implies that once a subgraph classifies as a redex, this situation will not change when other redexes are rewritten. Thus multiple redexes may be safely rewritten in parallel. The possibility of parallel graph rewriting is the reason that graph reduction is often considered as a suitable computational model for parallel computer architectures. One can interpret the program graph as a collection of potential parallel rewrite processes, where the graph expresses the communication and synchronisation needs between these processes [PEY87b].

Parallel computer architectures can be divided into architectures that support a global address space and those that do not. A global address space may be implemented on a shared memory or on a distributed memory. Parallel reduction machines are currently being implemented on both types of architectures [PEY87b, WAT87, HUD85]. Within the framework of the Dutch Parallel Reduction Machine Project [BAR87c], we have constructed an experimental machine [HER89] consisting of a collection of processors, each one equipped with a local memory. The processors are interconnected by a communication network, based on dual ported memories. It was decided not to support a global address space on this machine, because the hardware does not allow an efficient implementation.

There is a fundamental difficulty in implementing normal order graph reduction on parallel architectures that do not support a global address space. The problem is how to distribute the global graph representation over the available disjunct storage spaces. Although the interconnection of processors by dual ported memories provides a high communication bandwidth, our architecture still charges a significant communication cost to transport data from one memory to another. Because the cost to perform a single graph rewrite action can vary widely from fine grain to coarse grain it is not easy to decide which parts of the graph are worth being transported and reduced in parallel.

The approach that we have taken to implement normal order graph reduction on our parallel architecture is characterised by a compromise between pure graph reduction and pure string reduction. Within the local memories pure graph reduction is performed. When coarse grain sub-graphs are detected, they are transported and reduced in parallel on remote processors. This regime would be equivalent to string reduction, because transporting a sub-graph implies that it is copied to another storage space. To avoid the duplication of work, implied by pure copying, we have designed a special reduction strategy, which guarantees that the sub-graph to be transported is a primary redex (a primary redex contains no other redexes). For divide-and-conquer problems this reduction model has been demonstrated to yield good results on an experimental architecture [HAR88, VRE88]. A similar approach to parallel graph reduction with significant speed-up figures is reported in [MCB87].

The main subject of this paper is to demonstrate that apart from divide-and-conquer problems another class of programs, modelling process networks, can also be executed efficiently with our parallel reduction model. Writing a functional program as a network of stream processing functions is already discribed by Kahn [KAH74] in 1974. However, the purpose of the technique he introduced was purely theoretical, namely to describe the semantics of a set of communicating processes. Later this mathematical technique became of practical importance [WRA86, KEL89] , as efficient implementations of lazy functional languages emerged.

For a subset of application programs written as process networks we show that it is possible to eliminate all streams and therefore also cyclic stream connections by a program transformation that we call *communication lifting*. The transformed programs can be mapped onto our reduction model, which only supports strict operator parallelism. Such a mapping is not possible for the original application programs, because there is no way to express the required "pipe-line" parallelism of streams in a reduction model only based on strict operator parallelism.

In addition to communication lifting two other transformations, called the *sandwich*- and the *own*-transformation are informally described. These transformations are used to map programs resulting from communication lifting onto our parallel reduction model.

2 Job model

In our reduction model the programmer is required to annotate needed coarse grain sub-expressions, that we call jobs. Annotated jobs are evaluated by a special reduction strategy. Both annotation and strategy are effectuated by a rewrite rule called *sandwich*.

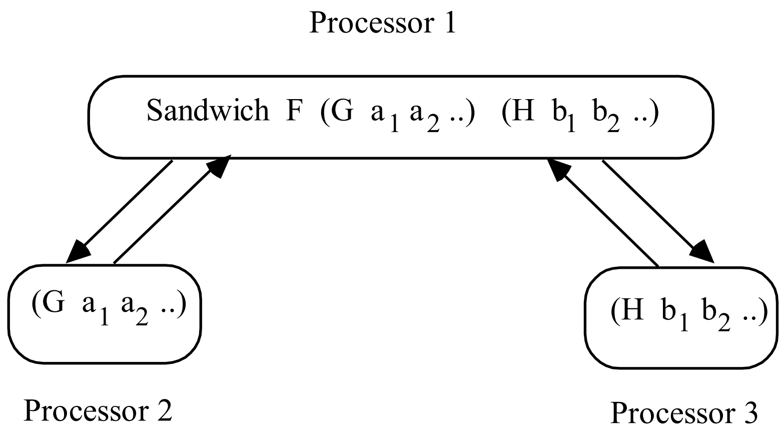


Figure 1: The sandwich annotation of two jobs

In figure 1 an occurrence of the sandwich rule is illustrated. The sandwich expression has the same meaning as *F (G a_1 a_2 ...) (H b_1 b_2 ...).* Function *F* must be strict in all its arguments. These arguments have to be coarse grain computations (jobs). The sandwich rewriting proceeds as follows: First the expressions $a_1\ a_2 \ldots$ and $b_1\ b_2 \ldots$ are reduced to normal form. This transforms the jobs $G\ a_1\ a_2 \ldots$ and $H\ b_1\ b_2 \ldots$ into primary redexes. Next these primary redexes are transported to remote processors, where they are reduced to normal form in parallel. Reduction of F is delayed until the results of its arguments are returned. Instead of transporting jobs to remote processors they may also be reduced locally, but a copy of the jobgraph will still be made.

To benefit from parallel execution, the application program must be transformed in such a way that a sandwich function can be inserted. This means that potential jobs have to be lifted to the same level, because they have to be annotated by one sandwich expression. Though this may seem an unnecessary constraint, it has the advantage that the implementation of the sandwich only has to synchronise once for all denounced jobs (one context switch). Also the availability of several jobs simultaneously, offers the possibility for better load-balancing decisions than when jobs are discovered one by one.

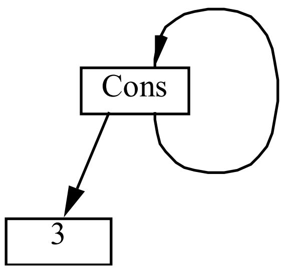
The main disadvantage of the presented job-model for parallel reduction is that only strict argument parallelism is supported. Application programs written as process networks do not seem to fit into our job-model, even though these application programs do exhibit a clear coarse grain structure (the processes).

Process networks are modeled in a functional language by tail recursive functions interconnected via streams. Streams are (infinite) lists that are produced and consumed element by element. If such functions (functional processes) would be distributed using the sandwich construct the special reduction strategy would normalise the streams, destroying the stream-property of element-wise production and consumption.

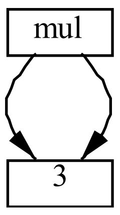
An additional difficulty arises when the interconnection pattern of a process network exhibits cyclic structures. Such programs cannot be split into separate independent jobs. In the remainder of this paper these problems will be examined and solutions will be presented that yield an efficient mapping of (cyclic) process networks onto the job model.

3 Graph rewriting

To discuss the graph-reduction behaviour of process networks we adopt a linear notation for graphs and graph rewriting from CLEAN [BRU87, BAR87b]. Each node in a CLEAN-graph contains a constructor symbol and is identified by a unique label. Constructor symbols start with a capital letter and labels are introduced by a post-fix colon. In principle each node may be supplied with a unique label, but often labels will be omitted when nodes are not shared. Two examples of a CLEAN-graph are given in figure 2. The examples illustrate the use of labels to denote sharing and cycles in a graph.



linear notation: n: Cons (3 n)



linear notation: Mul m: 3 m

Figure 2: Linear notation for two graphs

The first example is a graph representing the infinite list *3,3, ...* The second example denotes the squaring of *3*.

Graph rewrite rules consist of a *redex pattern* on the left-hand side and a *contractum pattern* on the right-hand side. Both patterns are CLEAN-graphs that may contain variables identifying arbitrary nodes. Consider for example the following rewrite rule:

```
Skip (Cons x (Cons y z))   →   Cons x z
```

The redex pattern contains three variables: *x*, *y* and *z*. Each of these variables is bound to a sub-graph during the matching of the redex pattern to the program graph. The top node of the matched sub-graph (which must contain the constructor symbol *Skip*) is then replaced by the graph specified by the contractum pattern. This means that the constructor symbol in the top node (*Skip*) is overwritten by *Cons* and that two pointers in the node are directed to the graphs matched by *x* and *z*.

Consider for example the reduction of the graph *(Skip (Cons 3 (Cons 4 5))*. The rewrite action according to the rule for *Skip* can be described with the linear notation in the following way:

```
k:  Skip (Cons l (Cons m n))      →    k: Cons l n
l:  3
m:  4
n:  5
```

The arrow (→) in the first line indicates that the node labeled *k* at the left hand side is destructively updated with the contents specified at the right hand side. The same reduction step can also be described in a graphical notation as shown in figure 3:

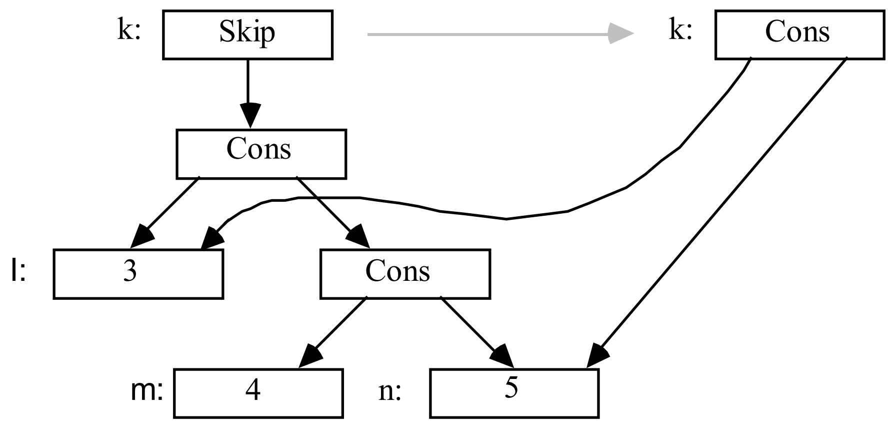


Figure 3: A graph rewrite

To present a case analysis of the reduction behaviour of process networks, we use the linear notation. The reduction steps can be easier presented using the linear notation than by drawing a sequence of pictures.

The sequential reduction order that is used in the examples of section 4 is determined by the *functional strategy*. This strategy performs reduction in much the same way as is usually the case in lazy functional languages. An operational description of this strategy can be found in [BRU87]. The strategy performs reduction in a left-most order. An important aspect of the strategy is that the process of matching a redex pattern may trigger recursive rewrite actions on the graph that is being matched.

4 Sequential graph reduction of process networks

In this section the graph reduction behaviour of cyclic process networks is illustrated with the aid of an example derived from a parallel simulation program for the tides in the North Sea. For the example an ad-hoc transformation is presented that results in an acyclic program, which can be mapped onto the job-based reduction model. Based on the ideas behind the ad-hoc transformation, a general transformation technique (communication lifting) is developed in section 5.

4.1 An example of a process network

As an example of a parallel functional program that is written as a process network, we consider a tidal model of the North Sea. In [VRE87] it is shown that such a functional program can be developed from the mathematical description by a number of systematical transformations. The use of processes connected by streams allows an elegant transformation of the sequential version of the tidal model into a coarse grain parallel version.

The program is based on a quantisation of physical reality on a two dimensional spatial grid, represented in the program by a matrix. To simulate the tides, the matrix is repeatedly updated, yielding a sequence of values that represent the physical state of the model at consecutive time steps. For the purpose of this section it is sufficient to consider a simplified model of the tidal simulation, which defines two coarse grains of parallelism. In figure 4 rounded boxes represent processes that consume and produce infinite lists (represented by arrows). The triangles at the bottom of figure 4 represent *Cons*-nodes that prefix the infinite lists by the initial values of both parts of the matrix (*mleft* and *mright*).

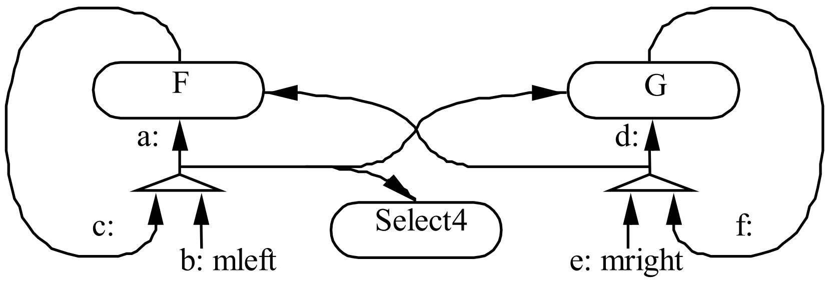


Figure 4: A simplified model of a parallel tidal simulation program

To obtain the coarse grains of parallelism the original matrix of the tidal model has been divided into two equal parts with the intention to calculate the updating of these parts in parallel (by processes *F* and *G* in figure 4). The process *Select4* is the main expression that produces the output of the simulation.

Although the program of figure 4 exhibits a coarse grain parallel structure, it does not fit into our job-model. The reason for this is the presence of global cyclic connections between the coarse grain parts of the program. Because a job is by definition a subgraph, cyclic structures always have to lie inside a job. Consequently the global cycles in figure 4 have to be part of one job that will be reduced sequentially.

Detailed measurements of the graph structure during sequential evaluation of the tidal model [HAR87], have shown that the global cycles disappear in an early stage of the execution. This means that although the initial graph is cyclic, most of the reduction work takes place in an acyclic graph, which might fit in our job-based reduction model. This observation has provided the incentive to look for a transformation that eliminates possible cycles from process networks. Before presenting this "communication lifting" transformation we show by an example why the cyclic structure only persists for such a short time during reduction.

The simplified program of figure 4 is sufficient to illustrate that the reduction behaviour of a cyclic process network can be split into two different phases. In the first phase a cyclic structure develops the spine of all infinite lists until the elements required by the main expression are produced. During the second phase the required elements are evaluated in an acyclic graph. As all computations of the physical model still remain to be done, the second phase requires much more time to reduce than the first.

In practice, a program like the tidal model, will be run to compute several snapshots of the tides during the evolution of the simulation. A main expression with this behaviour would be too complicated for a detailed presentation of the reduction process. Therefore we assume a simplified main expression (represented in figure 4 by *Select4*) that only selects the fourth element of the left stream (*a*). This will result in the computation of the left part of the state of the model after four simulated time slices. Representing the matrix-update operations inside *F* and *G* by *Mf* and *Mg*, the rewrite rules corresponding to figure 4 are the following:

```
Start                          →       Select4  a
                                       a:  Cons  b  c
                                       b:  mleft
                                       c:  F  a  d
                                       d:  Cons  e  f
                                       e:  mright
                                       f:   G  d  a

F  (Cons  x  xr) (Cons  y  yr)   →       Cons  b  c
                                       b:  Mf  x  y
                                       c:  F  xr  yr

G  (Cons  x  xr) (Cons  y  yr)   →       Cons  e  f
                                       e:  Mg  x  y
                                       f:   G  xr  yr
```

Select4 (Cons x_1 (Cons x_2 (Cons x_3 (Cons x_4 rest)))) → x_4

Figure 5: The rewrite rules corresponding to figure 4

The program is started with an initial graph consisting of the single node *Start*. Rewriting the start expression once, results in the graph *Select4 a*. In this graph the function *Select4* will select the fourth element of the list rooted at node *a*. The subgraphs *mleft* and *mright* represent respectively the left- and the right part of the matrix containing the initial state of the model.

To rewrite *Select4 a,* according to the functional strategy, means that the graph at node *a* will have to be reduced until it matches the required redex pattern of four *Cons* nodes. During this process the sub-graphs that match the variables x_1, x_2, x_3 and x_4 are not further reduced. We show the reduction steps, in functional order, that have to be performed to obtain the required matching of the graph rooted at *a*.

The initial graph in figure 6 is the graph rooted at node *a*, where all nodes have been provided with a subscript zero. To the right of the initial graph the first five reduction steps are illustrated, which are necessary to match a_0 to the four *Cons* nodes in the redex pattern of *Select4*. These five reduction steps constitute the first phase of the reduction process, during which the spines of the lists are developed.

A rewrite action is indicated by a dashed arrow, and the rewritten top node plus all new nodes introduced by the rule are listed directly to the right of the arrow. The new nodes have to be provided with unique labels. The identifiers for new node labels are derived from the identifier in the rule plus a subscript. When a redex is rewritten the node label is not repeated at the right side of the arrow, to stress the fact that it is the same node as the one to the left of the arrow. For instance c_0*: F* a_0 d_0 $\rightarrow$ *Cons* b_1 c_1 means that the node *F* at c_0 is rewritten to a *Cons* node and its pointers are redirected to the (new) nodes at b_1 and c_1.

<u>initial graph</u>	<u>steps 1 and 2</u>	<u>steps 3 and 4</u>	<u>step 5</u>
a_0: Cons b_0 c_0			
b_0: *mleft*			
c_0: F a_0 d_0 $\rightarrow$	Cons b_1 c_1		
	b_1: Mf b_0 e_0		
d_0: Cons e_0 f_0	c_1: F c_0 f_0 $\rightarrow$	Cons b_2 c_2	
e_0: *mright*		b_2: Mf b_1 e_1	
f_0: G d_0 a_0 $\rightarrow$	Cons e_1 f_1	c_2: F c_1 f_1 $\rightarrow$	Cons b_3 c_3
	e_1: Mg e_0 b_0		b_3: Mf b_2 e_2
	f_1: G f_0 c_0 $\rightarrow$	Cons e_2 f_2	c_3: F c_2 f_2
		e_2: Mg e_1 b_1	
		f_2: G f_1 c_1	

Figure 6: Rewriting the program of figure 5 in linear notation.

The initial graph contains two rewritable subgraphs at the nodes c_0 and f_0. The node at c_0 is an occurrence of the rule for *F*, because both its arguments (a_0, d_0) are applications of the *Cons* constructor. For the same reason the node at f_0 is an occurrence of the rule for *G*.

The functional strategy will try to develop a spine of four *Cons* nodes starting from node a_0, to match the redex pattern of the *select4*-rule. The node at a_0 is already a *Cons* node, so the strategy first rewrites the redex at c_0 to obtain the second *Cons* node needed for the match. After rewriting c_0, the next redex that has to yield a *Cons* node is c_1. However, during the matching of the *F*-rule to c_1, a recursive rewrite of the redex at f_0 will occur, to convert this node (f_0) into a *Cons* node, as required by the pattern of *F*. The rest of the reduction steps follows the same order as the first three steps.

The graph rooted at a_0 that is obtained in figure 6 after five reduction steps is redrawn in figure 7, using a graphical notation, to illustrate some characteristics of this graph. Three different structures can be distinguished in figure 7. The first is the beginning of the spine of the infinite list at a_0, (the nodes a_0, c_0, c_1 and c_2). The second structure is an acyclic graph, specifying a communication pattern between the matrix update operations *Mf* and *Mg*. The third part of the graph is a cyclic structure involving *F* and *G*, that forms a generator for both the spine and the communication pattern.

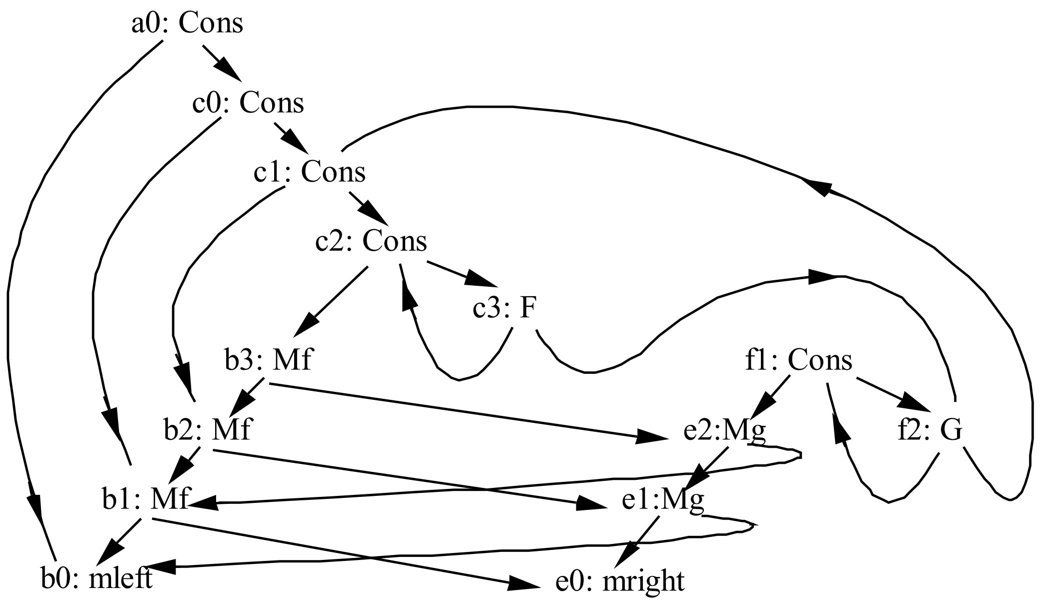


Figure 7: The program graph after five reduction steps

Because the spine at a_0 consists of four *Cons* nodes, the rewrite rule for *Select4* can now be applied. The result of this rewrite is shown in figure 8.

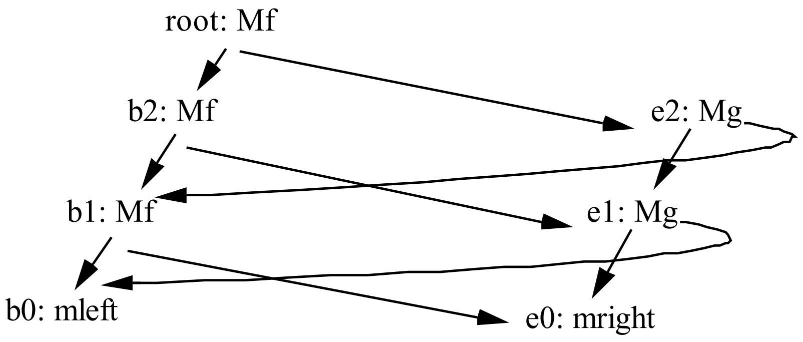


Figure 8: The second phase of the reduction process

The generator has disappeared and the cycles with it. Just the communication pattern between the matrix-update functions is left behind. The graph, however, still represents a considerable amount of work. The functions *Mf* and *Mg* contain quite complex updating actions on the matrix of the tidal model. Five applications of these functions remain to be reduced. The number of reduction steps involved dwarves the six steps that were required to obtain the graph of figure 8. The graph in figure 8 also suggests a distribution of the reduction work over two processors, where the functions *Mf* are evaluated on one processor and the functions *Mg* on the other.

4.2 Communication lifting of the example program

The graph reduction example in the previous section illustrates that cyclic structures in a process network, appear to be generators for a possibly complex but acyclic communication pattern between calculations contained in the processes. The mechanism of lazy evaluation first develops the communication pattern and then discards the generators before evaluating the major part of the computations. The question arises whether it is possible to construct an acyclic generator for these patterns in a systematical way. A general method to obtain an acyclic program that is equivalent to a process network is presented in the next section. The idea behind the method is illustrated in figure 9:

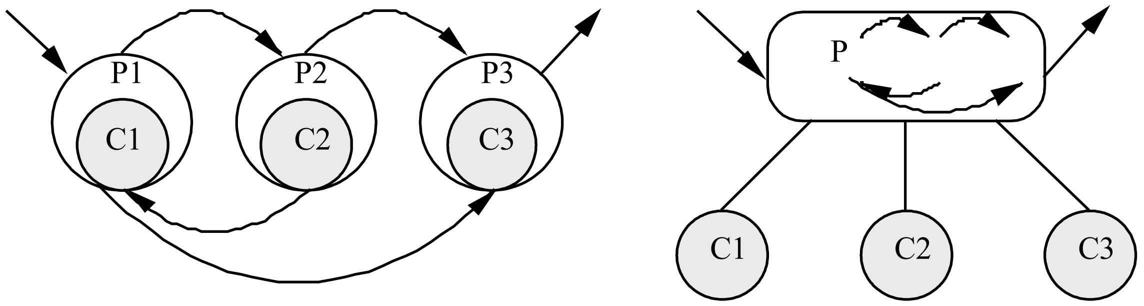


Figure 9: Communication lifting

The left hand side of figure 9 shows three processes *P1*, *P2* and *P3*. The arrows between the processes represent streams that model the communication between processes. One can think of a process as being composed of two parts (the white and shaded areas in figure 9). One part (white) deals with the incoming and outgoing streams. Data is selected from the input streams and passed to a second part (shaded) where the actual calculations are performed. The obtained results are combined into the output stream by the first part again. Under certain conditions it is possible to isolate the communication parts of *P1*, *P2* and *P3* and combine them into a single process *P*. The calculation parts *C1*, *C2* and *C3* remain unchanged. They are called by *P* as normal function applications involving no streams. Because the communication that originally occurs between *P1*, *P2* and *P3*, now takes place inside a single process on a higher level, we have called the transformation method *communication lifting*. The processes *P1*, *P2* and *P3* have to meet certain synchronisation constraints before communication lifting can be applied. Processes that fulfil these conditions are called *synchronous processes*. The definition of synchronous processes and the lifting transformation are described in the next section.

In the remainder of this section we present the acyclic program for the simplified tidal model (see figure 10) that is obtained by communication lifting of the cyclic version (of figure 5). We show that reduction of this program produces exactly the same computational graph as with the cyclic version (namely the graph of figure 8). This demonstrates that communication lifting indeed produces an acyclic generator for the program of figure 5.

```
Start            →    Select4  a
                      a:  Proc  b  c
                      b:  mleft
                      c:  mright
```

Proc x y	→	Cons x a
		a: Proc b c
		b: Mf x y
		c: Mg y x

Figure 10: The acyclic rules corresponding to figure 4

Although the lifted version looks simpler than the original cyclic generator (figure 5), this is not the case for larger programs like the complete tidal model. When communication patterns in these type of applications grow more complicated, the lifted version becomes difficult to understand. This is because the knowledge of how processes communicate is hidden in the order of the many parameters that have to be passed to the single recursive function that is obtained by communication lifting (like *x* and *y* in *Proc*).

Like in the previous section, rewriting of the initial graph *Start* produces the graph *Select4 a* . Four subsequent reduction steps generate the spine of *a* until the four *Cons* nodes required by *Select4* are developed:

a_0: Proc b_0 c_0	→	Cons b_0 a_1						
b_0: *mleft*		a_1: Proc b_1 c_1	→	Cons b_1 a_2				
c_0: *mright*		b_1: Mf b_0 c_0		a_2: Proc b_2 c_2	→	Cons b_2 a_3		
		c_1: Mg c_0 b_0		b_2: Mf b_1 c_1		a_3: Proc b_3 c_3	→	Cons b_3 a_4
				c_2: Mg c_1 b_1		b_3: Mf b_2 c_2		a_4: Proc b_4 c_4
						c_3: Mg c_2 b_2		b_4: Mf b_3 c_3
								c_4: Mg c_3 b_3

The next reduction step is the application of the rule for *Select4*. It rewrites to the graph rooted at *b_3*, the fourth element of the spine. The reader may verify that the graph below node *b_3* is homomorph with the graph of figure 8.

Using the sandwich rule of section 2, the lifted program of figure 10 can be mapped onto the job-based parallel reduction model. The resulting program is shown in figure 11.

Start	→	Select4 (Proc *mleft* *mright*)
Proc x y	→	Cons x (Sandwich Proc (Mf x y) (Mg y x))

Figure 11: The sandwich version of the simplified tidal model

The *Sandwich* strategy first nomalises *x* and *y*, then dispatches the jobs *Mf x y* and *Mg y x* for parallel evaluation and finally combines the job-results into a new application of *Proc*.

A transformation into a parallel sandwich version is impossible for the original program of figure 5, because the right hand side of the *Start*-rule is a cyclic graph.

5 Communication lifting

When the right hand side of a rewrite rule contains several (needed) coarse grain subgraphs, it is possible to obtain a parallel version of this rule using the sandwich function of section 2. However, when these coarse grain subgraphs are contained within a cyclic structure the method fails, because the arguments of the sandwhich rule have to be independent function applications (jobs). The previous section suggests that in some cases a cyclic rule (i.e. the right hand side graph contains a cycle) can be transformed into an acyclic rule that produces the same result. The acylic rule can be transformed into a parallel version using the sandwich function.

In this section we present the *communication lifting* transformation that can be used to obtain an acyclic rule in case the (cyclic) right hand side graph is a *synchronous process network*. We advocate a programming methodology in which an application program is first developed using streams and synchronous processes, because this approaches physical reality and our way of thinking about physical problems. Next, the program is transformed by communication lifting. In the resulting program coarse grain subexpressions are annotated by the programmer, yielding a version of the program that can be efficiently executed on our parallel reduction model that only supports coarse grain strict argument parallelism.

Apart from making a stream based program suitable for strict argument parallelism, communication lifting can also be used to transform a (cyclic) graph of communicating fine-grain processes into a single application of one (acyclic) coarse-grain process.

In this section we describe communication lifting as a set of transformation rules operating on a CLEAN-graph and a CLEAN-rule-set. Throughout the section we use the program of figure 5 to illustrate the formal description of the transformation rules.

A *synchronous process* is defined as a function that operates according to one of the following models (henceforward the dot is used as an infix notation for *Cons*):

$$\mathrm{F}\ \ \mathrm{s}\ \ (\mathrm{x}_1 \,.\, \mathrm{xr}_1) \ldots (\mathrm{x}_n \,.\, \mathrm{xr}_n) \ \rightarrow\ (\mathrm{g}\ \ \mathrm{s}\ \ \mathrm{x}_1 \ldots\ \mathrm{x}_n)\ .\ \mathrm{F}\ \ (\mathrm{f}\ \ \mathrm{s}\ \ \mathrm{x}_1 \ldots\ \mathrm{x}_n)\ \ \mathrm{xr}_1 \ldots .\ \mathrm{xr}_n \qquad \text{(S1)}$$

where	s	is the state of the process F
	$(\mathrm{x}_i \,.\, \mathrm{xr}_i)$	are input streams to F
	g	is a function that computes the next output element of F
	f	is a function that computes the next state of F

In *(S1)* the process *F* rewrites to a pair, consisting of the application of an output-generating function *g*, followed by a recursive invocation of *F*. In the recursive call the new state of the process is computed by a state-transforming function *f*. The essential property of a synchronous process is that it consumes one element from all the input streams to produce one element of the output stream. This does not mean that the consumed elements are all needed in the computation of the output element. Input elements might be skipped during the execution of a synchronous process.

Also processes without a state paramenter, according to model *(S2)* are considered as synchronous processes:

$$F\ (x_1 \,.\, xr_1) \ldots (x_n \,.\, xr_n) \rightarrow (g\ x_1 \ldots\ x_n) \,.\, F\ xr_1 \ldots.\ xr_n \qquad \text{(S2)}$$

Although model *(S1)* and *(S2)* seem rather restricted, they can be used to describe a large class of application programs. In principle all grid-based computations occurring in mathematical models, image processing, computer graphics, VLSI design, discrete simulations etc, can be modelled as a collection of synchronous processes. Using communication lifting and sandwich transformations, grain-size and parallelism can be controlled in a general, application independent way. As an example we show in section 6 the transformation of the tidal model and a simple simulation of digital hardware.

5.1 Application requirements

The application program has to meet four requirements (*S3-S6*) before communication lifting can be applied. These requirements are specified §5.1.1 through §5.1.4.

Communication lifting is applied to a set of rules *FN* and a possibly cyclic subgraph *GR*. This subgraph is (part of) the right hand side of a rewrite rule.

5.1.1 Syntactical form of *FN*

The syntactical form of the definitions in *FN* is defined by *(S3)*. There are *m* processes in *(S3)*, from which *n* processes are according to model *(S1)* and the rest according to model *(S2)*

$$\begin{array}{lll} FN:: & F_i\ s_i\ (x_{i1} \,.\, xr_{i1})\ (x_{i2} \,.\, xr_{i2}) \ldots & \text{(S3)} \\ & \rightarrow \quad (g_i\ s_i\ x_{i1}\ x_{i2} \ldots) \,.\, F_i\ (f_i\ s_i\ x_{i1}\ x_{i2} \ldots)\ xr_{i1}\ xr_{i2} \,.. \quad (i = 1..\ n) & \\ & & \\ & F_i\ (x_{i1} \,.\, xr_{i1})\ (x_{i2} \,.\, xr_{i2}) \ldots & \\ & \rightarrow \quad (g_i\ x_{i1}\ x_{i2} \ldots) \,.\, F_i\ xr_{i1}\ xr_{i2} \,.. \qquad (i = n{+}1..\ m) & \end{array}$$

In a rule each variable identifier must be different from variable identifiers occurring in other rules. This is because the communication lifting transformation performs textual substitutions that are global with respect to all definitions in *FN*. In *(S3)* uniqueness has been achieved by a double subscript for the head- and tail variable of each stream. The first subscript indicates the rule number, whereas the second subscript enumerates the stream variables in each rule.

5.1.2 Syntactical form of *GR*

The interconnections between the processes F_i in *FN* are specified by *GR*. This subgraph has to consist of labeled applications of the following form:

$$\begin{array}{llll} GR:: & a_i : F_i\ t_i\ b_{i1}\ b_{i2}\ \ldots & (i = 1..\ n) & \text{(S4)} \\ & a_i : F_i\ b_{i1}\ b_{i2}\ \ldots & (i = n{+}1..\ m) & \end{array}$$

The actual stream arguments b_{ij} in *GR* have to be node labels. We require that each stream is produced by a labeled application of a synchronous process. Note the use of s_i and t_i for the formal, respectively the actual state-argument of F_i. Similarly x_{ij} and b_{ij} denote the formal, respectively the actual stream-arguments of F_i.

In *GR* a connection exists between two processes F_i and F_j if $\exists k \mid b_{ik} = a_j$. In that case we say that the output stream of F_j is connected to the k^{th} input of process F_i. The k^{th} input of process F_i is called unconnected if $\forall j \mid b_{ik} \neq a_j$. The values of unconnected inputs are determined by the right hand side graph from which *GR* has been selected.

example:

As an example we derive the rule set *FN* and the graph *GR* corresponding to the cyclic right hand side of the *Start*-rule in figure 5. In a first step we neglect the syntactical form and merely identify *FN* and *GR* (see figure 12). In a second step (figure 13) we bring the definitions in the required syntactical form.

```
FN::
F  (x . xr)  (y . yr)  →  Mf  x  y  .  F  xr  yr
G  (x . xr)  (y . yr)  →  Mg  x  y  .  G  xr  yr
```

```
GR::
a: Cons  b  c ;   c: F  a  d
d: Cons  e  f ;   f: G  d  a
```

Figure 12a: FN and GR of the *Start*-rule of figure 5

```
Start            →        Select4  a
                          b:  mleft
                          e:  mright
```

Figure 12b: The remaining part of the *Start*-rule

In *GR* of figure 12a the labels *b* and *e* are unconnected. They are defined in the remaining part of the *Start* rule (see figure 12b) and will become the external inputs for the transformed program. The graph *GR* does not yet satisfy the constraints outlined above because the two streams at label *a* and *b* are not produced by a synchronous process. However, we can replace the applications of *Cons* by applications of a *Cons*-process, which we define as:

```
Cp  s  (x . xr)   →       s  .  Cp  x  xr
```

The definition of *Cp* complies with the model of a synchronous process and if *x* is a list it holds that *Cp s x = Cons s x*.

Figure 13 shows the example program again, where *Cons* has been replaced by the *Cons*-process and where all rule definitions in *FN* and applications in *GR* have been subscripted as prescribed by *(S3)* and *(S4)*:

```
FN::  Cp1 s1 (x1 . xr1)           →   s1 . Cp1 x1 xr1
      Cp2 s2 (x2 . xr2)           →   s2 . Cp2 x2 xr2
      F3 (x3 . xr3) (y3 . yr3)    →   Mf x3 y3 . F3 xr3 yr3
      G4 (x4 . xr4) (y4 . yr4)    →   Mg x4 y4 . G4 xr4 yr4

GR::  a1: Cp1 b c3 ;       c3: F3 a1 d2
      d2: Cp2 e f4 ;       f4: G4 d2 a1
```

Figure 13: The simplified tidal model satisfying the constraints for FN and GR

5.1.3 Correspondence between *GR* and *FN*

There should be a one-to-one correspondence between application nodes in *GR* and rule definitions in *FN*:

$$\forall i \in 1..m \quad | \quad (a_i\!: F_i \ldots.) \in GR \quad \Leftrightarrow \quad F_i \in FN \qquad \text{(S5)}$$

If *GR* contains multiple applications of the same function, then copies of the corresponding function definition (but with a different subscript) have to be added to *FN*.

example:

Note that in figure 13 both *Cons* applications have been replaced by applications of two distinct functions *Cp_1* and *Cp_2*, whereas the definitions for *Cp_1* and *Cp_2* are identical (the *Cons*-process). Communication lifting as described in the next sections, requires that all nodes in *GR* are applications of distinct functions in *FN*. The duplication of function definitions caused by this requirement has no serious consequences because it is temporary. Later *FN* will be replaced by a single function definition after communication lifting.

5.1.4 One output application node

In *GR* one labeled application *(a_{out} : F_{out} s_{out} $b_{out,1}$ $b_{out,2}$ ...)* has to generate the output stream of the group of processes described by *FN* and *GR*:

$$\exists!\, out \in 1..m \quad | \quad a_{out} : F_{out}\, s_{out}\, b_{out,1}\, b_{out,2} \ldots \text{ is the output of GR} \qquad \text{(S6)}$$

The output stream of *GR* is determined by the right hand side graph of the rule from which *GR* has been selected. If more streams represent the output of the group then an extra process has to be added to the group that merges the output streams into one stream.

example:

Figure 12b shows that variable *a* in the expression *(Select4 a)* is no longer defined after *GR* has been isolated for transformation. Therefore node *a* in *GR* (which is node *a_1* in figure 13) becomes the output stream. Thus, for the example program: $F_{out} = Cp_1$ and $a_{out} = a_1$.

5.2 The description of communication lifting

Assuming that a rule set *FN* and a cyclic subgraph *GR* satisfy the constraints outlined in the previous section, communication lifting can be defined as a set of syntactical transformations to be applied to *FN* and *GR*. The result is that *FN* will be replaced by a single acyclic function definition *G*, whereas *GR* will be replaced by a single application of *G*. We describe the transformation in six steps *T1* to *T6* :

5.2.1 The calculation of the communication matrix - step T1

Instead of directly referring to stream connections in *GR*, the description of communication lifting uses a two dimensional matrix *C*, which we call the communication matrix. All connections in *GR* are represented by *C*. An informal definition of *C* is (referring to *(S3)* and *(S4)*):

$$C_{ij} = \{ x_{jk} \mid \text{the } k^{th} \text{ input of } F_j \text{ is connected to the output of } F_i \}$$

When the k^{th} input (actual argument) of F_j is connected to the output of F_i the matrix element C_{ij} contains the head-variable x_{jk} of the k^{th} formal argument of F_j. The matrix element C_{ij} is a set because several inputs of F_j may be connected to the output of F_i.

A formal specification for *C* is given by *T1* (referring to *(S3)* and *(S4)*):

```
-T1   ∀ i, j ∈ 1.. m
          ∀ k ∈ 1 ..  the number of stream arguments of $F_j$
          iff   GR contains two nodes $a_i$ , $a_j$ such that:
                      $\mathbf{a_i}$: $F_i$ $t_i$ $b_{i1}$ $b_{i2}$ ...
                      $a_j$: $F_j$ $t_j$ $b_{j1}$ ... $b_{j,k-1}$ $\mathbf{a_i}$ $b_{j,k+1}$ ....
                and   the definition of $F_j$ in FN is:
                      $F_j$ $s_j$ ($x_{j1}$ . $xr_{j1}$) ... ($\mathbf{x_{jk}}$ . $xr_{jk}$) ...   →   $g_j$ ... . $F_j$ ...
          then  $\mathbf{x_{jk}} \in C_{ij}$
```

The specification *(T1)* assumes that F_i and F_j syntactically comply with model *(S1)*. Slight variants of *(T1)* have to be used when rules of model *(S2)* are involved.

example:

When *(T1)* is applied to the example program of figure 13, we have e.g.: $C_{31} = \{x_1\}$ because of the connection between F_3 and Cp_1 in *GR* (connection in bold face):

```
$a_1$: $Cp_1$ b $\mathbf{c_3}$
$\mathbf{c_3}$: $F_3$ $a_1$ $d_2$
```

and the corresponding definition of Cp_1 in *FN* :

$$Cp_1\ s_1\ (\mathbf{x_1} . xr_1) \quad \rightarrow \quad s_1 . Cp_1\ x_1\ xr_1$$

The output of the application c_3: $F_3\ a_1\ d_2$ is second actual argument of CP_1. This argument corresponds to the second formal argument: $(x_1 . xr_1)$ in the definition of Cp_1. Therefore, according to *(T1)*, x_1 is a member of C_{31}.

Similarly, we can derive the other non-empty elements of C :

$$C_{3,1} = \{x_1\},\ C_{4,2} = \{x_2\},\ C_{1,3} = \{x_3\},\ C_{1,4} = \{y_4\},\ C_{2,3} = \{y_3\},\ C_{2,4} = \{x_4\}$$

5.2.2 The transformation of FN - steps T2 through T4

The communication lifting transformation finds a single rewrite rule *G*, which has the same behaviour as the connected set of processes defined by *FN* and *GR*. The rewrite rule for *G* is constructed by applying textual transformations to the following rule-pattern:

G State Input-streams → Output . G New-state Stream-tails

The skeleton of *G* contains five meta-variables: *State*, *Input-streams*, *Output*, *New-state* and *Stream-tails*. We use the notation $G[E_1/E_2]$ to denote that in the rule for *G* each occurrence of E_2 is textually replaced by E_1.

-T2 *State* is replaced by a tuple consisting of the formal state arguments of the processes F_1 to F_n in *FN* (in *(S3)* the functions F_1 to F_n are according to model *(S1)* and have formal state arguments):

G [$(s_1 . s_2 . \dots . s_n)$ / State]

New-state is replaced by a tuple consisting of the state-transforming expressions of the processes F_1 to F_n in *FN* :

G [$(f_1\ s_1\ x_{11}\ x_{12}\ \dots) . (f_2\ s_2\ x_{21}\ x_{22}\ \dots) . \dots . (f_n\ s_n\ x_{n1}\ x_{n2}\ \dots)$ / New-state]

In *FN* there is one process F_{out} that produces the output of the group (see §5.1.4). *Output* is replaced by the output generating expression of process F_{out}:

G [$g_{out}\ s_{out}\ x_{out,1}\ x_{out,2}\ \dots$ / Output]

-T3 In both expressions for *New-state* and *Output*, all variables that are involved in communication have to be replaced according to the following rule:

$\forall i, j \in 1..m$ and $\forall k$ 1 .. the number of stream arguments of F_j

if $x_{j\,k} \in C_{i\,j}$

then G [$(g_i\ s_i\ x_{i1}\ x_{i2}\ \dots)$ / $x_{j\,k}$]

In other words: each variable in *New-state* and *Output* that corresponds to a stream connection between process F_i and F_j (i.e. $x_{jk} \in C_{ij}$) is replaced by the output generating expression taken from the definition of process F_i in *FN*.

The replacements described by *T3* have to applied repeatedly until no more variables can be replaced. Common subexpressions that may arise during the substitutions must be replaced by a single labeled expression. In particular cyclic connections would otherwise result in an infinite sequence of textual substitutions.

-T4 Finally the meta-variables *Input-streams* and *Stream-tails* are to be replaced by a list of arguments. These lists contain the stream-arguments in *FN* that correspond to unconnected inputs in *GR*: The order of the elements in the argument lists has to be the same for both replacements.

G [(argument-list of all ($x_{j\,k}$. $xr_{j\,k}$) such that $x_{j\,k} \notin C_{i\,j}$) / Input-streams]
G [(argument-list of all $xr_{j\,k}$ such that $x_{j\,k} \notin C_{i\,j}$) / Stream-tails]

where i, j 1.. m and k 1.. number of stream arguments of F_j

example:

To illustrate the rules T2-T4 we construct the function *G* for the example program of figure 13, for which we have already derived the communication matrix *C*.

T2: In figure 13 only Cp_1 and Cp_2 are according to model *(S1)*. Thus, referring to *(S3)*: $n = 2$ and $m = 4$, which yields the following replacements following *(T2)*:

G [(s_1 . s_2) / State]
G [(x_1 . x_2) / New-state]

In §5.1.4 we have derived that $F_{out} = Cp_1$. The output generating expression of Cp_1 is just the variable s_1, so *Output* is replaced by s_1.

G [s_1 / Output]

Applying the replacements of step *(T2)* to the skeleton of *G,* we obtain the following intermediate result:

G (s_1 . s_2) Input-streams → s_1 . G (x_1 . x_2) Stream-tails

T3: The next step in the transformation *(T3)* performs a number of substitutions on the variables in the expressions for *Output* and *New-state* that have been obtained so far. In the example only the two variables in *New-state* have to be replaced. Because $x_1 \in C_{31}$, rule *(T3)* states that x_1 has to be replaced by the output generating expression of F_3, which is $Mf\ x_3\ y_3$. Similarly x_2 is replaced by $Mg\ x_4\ y_4$:

G [((Mf x_3 y_3) . (Mg x_4 y_4)) / (x_1 . x_2)]

However, the variables x_3, y_3, x_4 and y_4 all occur in the communication matrix *C.* So according to *(T3)*, they have to be replaced again. For example because $x_3 \in C_{13}$ and the output generating expression of Cp_1 is s_1, we have to replace x_3 by s_1. Performing similar substitutions for y_3, x_4 and y_4, we obtain:

G (s_1 . s_2) Input-streams → s_1 . G ((Mf s_1 s_2) . (Mg s_2 s_1)) Stream-tails

T4: In the next step of communication lifting *(T4)* the meta-variables *Input-streams* and *Stream-tails* are replaced. To perform *T4* we have to find the set of all formal stream arguments *(x_{jk} . xr_{jk})* in *FN* that correspond to unconnected (actual) streams in *GR*, i.e. $x_{jk} \notin C_{ij}$. In case of the example program this set turns out to be empty (see section 6 for an example where the set is non-empty). Therefore the meta-variables *Input-streams* and *Stream-tails* are replaced by empty argument lists, which results in the final definition for *G*:

New-FN:: G (s_1 . s_2) $\rightarrow$ s_1 . G ((Mf s_1 s_2) . (Mg s_2 s_1))

We have introduced the name *New-FN* to indicate that the definition obtained for *G* replaces all definitions of *FN*.

5.2.3 The transformation of GR - steps T5 and T6

Communication lifting as described so far, replaces the set of function definitions *FN* by a single definition *G*. What remains to be done is to transform the subgraph *GR* into a single application of *G*.

In *GR* there is exactly one node (a_{out}) that produces the output stream of the subgraph. We replace this node by an application that is constructed from the left hand side of the definition of *G* obtained in the previous section. The construction replaces each formal argument of *G* by actual arguments taken from *GR*, according to the following rules *(T5)* and *(T6)*:

–T5 $\forall$ k $\in$ 1..n :

$$\left.\begin{array}{l} New\text{-}\ FN :: G(s_1 \ldots \mathbf{s_k} \ldots s_n) \ldots \rightarrow \ \ldots \\ GR :: a_{\mathbf{k}} : F_k \mathbf{t_k} b_{k1} b_{k2} \ldots \end{array}\right\} \Rightarrow \text{New-GR:: } a_{out}\text{: G } (s_1 \ldots \mathbf{t_k} \ldots s_n) \ldots$$

–T6 ($\forall$ i, j $\in$ 1..m and $\forall$ k $\in$ 1..number of stream arguments of F_j) | $x_{jk} \notin C_{ij}$:

$$\left.\begin{array}{l} New\text{-}\ FN :: G(\ldots)\ldots(\mathbf{x_{jk}} . \mathbf{xr_{jk}})\ldots \rightarrow \ \ldots \\ GR :: a_{\mathbf{j}} : F_j t_j b_{j1} \ldots \mathbf{b_{jk}} \ldots \end{array}\right\} \Rightarrow \text{New-GR:: } a_{out}\text{: G } (\ldots) \ldots \mathbf{b_{jk}} \ldots$$

Rule *(T5)* describes how in *New-GR* each formal state variable s_k of the definition of *G* is replaced by an actual state argument t_k from *GR*. The bold face letters help to identify which variables are replaced and where to find the actual argument. As before, the numbers *n* and *m* in the quantisation are defined in *(S3)*. Rule *(T6)* specifies how each formal input-stream of *G* is replaced by an actual stream argument from *GR*. Finally, the graph *New-GR*, consisting of the single node a_{out}, replaces the old graph *GR*.

example:

To complete the transformation of the example program we apply rule *T5* to the subgraph *GR* of figure 13. We know already that $a_{out} = a_1$ for the example. This node will be replaced by an application of *G*. To construct the application of *G* we take the left hand side of its definition: *G (s_1 . s_2)* and replace the state variables by actual arguments in *GR*. Applying rule *(T5)* in two steps yields:

$$\left.\begin{array}{l} New\text{-}\ FN :: G(\mathbf{s}_1.s_2)... \to\ ... \\ GR :: a_1 : Cp_1\mathbf{b}c_3 \\ d_2 : Cp_2ef_4 \end{array}\right\} \Rightarrow \quad \text{New-GR::} \quad a_1\text{: G } (\mathbf{b}\ .\ s_2)$$

$$\left.\begin{array}{l} New\text{-}\ FN :: G(s_1.\mathbf{s}_2)... \to\ ... \\ GR :: a_1 : Cp_1bc_3 \\ d_2 : Cp_2\mathbf{e}f_4 \end{array}\right\} \Rightarrow \quad \text{New-GR::} \quad a_1\text{: G } (b\ .\ \mathbf{e})$$

Rule *(T6)* has no effect in the example, because *G* has no formal input streams (i.e. the quantisation in *(T6)* is void). The application that replaces *GR* therefore remains:

New-GR:: a_1: G (b . e)

The results obtained for *New-FN* and *New-GR* can now be recombined with the remaining part of the *Start-rule* of figure 12b. This yields the final result of communication lifting applied to the example program (of figure 5):

```
Start   →     Select4 a
              a: G (b . e)
              b: mleft
              e: mright
G (s1 . s2)  →  s1 . G (( Mf s1 s2 ) . ( Mg s2 s1 ))
```

Figure 14: The final result of communication lifting of the example program of figure 5.

5.2.4 Correctness of the communication lifting

Apart from the pairing of the state variables, the names used for these variables and some extra labels, the program of figure 14 is identical to the solution presented in figure 10. After a few reduction steps the program of figure 14 reduces to the same graph as shown in figure 8 and thus computes the same output as the original program of figure 5.

It is difficult to prove the correctness of communication lifting, because the contents of the communication matrix is in general unknown. For a given program the correctness of communication lifting can be proven by showing that the output stream of the transformed program is identical to the original output stream (there is always a single output stream, see §5.1.4). Such a proof can be constructed by induction on the elements of both streams. In the example program one has to prove that *a: G (b . e)* of figure 14 is identical to *a: Cons b c* of figure 5.

5.3 Sandwich transformations

A synchronous process *G* that is the result of communication lifting of a group of processes *FN* and a subgraph *GR*, can be mapped onto the job-based parallel reduction model. The output-generating expression and the state-transforming expression of *G* are well suited for *sandwich* annotation. Both expressions only contain applications of the functions f_i and g_i of the original processes in *FN*. In principle all needed applications of f_i and g_i are possible candidates to become part of a *sandwich* expression. The programmer has to select the coarse grain needed expressions and isolate them by a transformation. To give an impression of such transformations we use the function *G* of figure 15. This is an example of a function generated by communication lifting that is simple, but sufficiently complex to illustrate relevant aspects of the *sandwich* transformation and the *retention* of results (the latter subject is discussed in the next subsection):

```
G (s1 . s2)    →    a . G ((f1 s1 b) . (f2 s2 a))
                    a: g1 s1
                    b: g2 s2
```

Figure 15: A typical function generated by communication lifting

First we show how the *sandwich* rule has to be incorporated. Depending on the grain size of the various calculations there are three possible transformations:

```
G (s1 . s2)    →    a . G (Sandwich Cons (f1 s1 b) (f2 s2 a))     (SW1)
                    a: Head c
                    b: Tail c
                    c: Sandwich Cons (g1 s1) (g2 s2)

G (s1 . s2)    →    a . G ((f1 s1 b) . (f2 s2 a))                 (SW2)
                    a: Head c
                    b: Tail c
                    c: Sandwich Cons (g1 s1) (g2 s2)

G (s1 . s2)    →    a . G (Sandwich Cons (f1 s1 b) (f2 s2 a))     (SW3)
                    a: g1 s1
                    b: g2 s2
```

The first transformation is appropriate when the grain size of both the output- and the state calculations is sufficiently large to justify parallel evaluation. The *sandwich* annotation always requires a main function operating on the received job-results. We use the *Cons* constructor for this purpose. This means that if one of the paired (*Cons*-ed) results is required, it has to be selected with the *Head*- or *Tail*-function. In *(SW1)* this occurs at node *a* and *b* to select the results of $(g_1\ s_1)$ and $(g_2\ s_2)$. Unpairing the result of the other sandwich expression is not necessary, because a *Cons*-ed pair happens to be needed as the argument to *G*.

If the computation of the next state does not outweigh the communication cost involved in the transmission of the jobs $(f_1\ s_1\ \ b)$ and $(f_2\ s_2\ \ a)$, these functions need to be reduced sequentially, as is illustrated by transformation *(SW2)*.

The third transformation applies when most of the work is involved in evaluating the next state. This is generally the case for loosely coupled processes, like the tidal model and the hardware simulation of section 6.

In most of these applications the state is represented by a large data structure. The parallel jobs as they are annotated in *(SW3)* contain the states s_1 and s_2. Each time when both jobs are dispatched for parallel evaluation, all data representing the states is also transmitted. This may cause an unacceptable amount of communication.

Inspired by the execution pattern of *(SW3)* we have designed an extra annotation that can be used in conjunction with the *Sandwich* annotation, to avoid the repeated transmission of states. The annotation has been called the *own*-annotation because it causes processors to retain their (own) result. The *own*-annotation is described in detail in [VRE88]. Here we only give a short explanation and show how program *(SW3)* has to be transformed again, to profit from the retention of function results, caused by the *own*-annotation.

5.4 The retention of results

Figure 16 gives an impression of the effect of the own-annotation, implemented on a local memory architecture consisting of two processors. Suppose that during the evaluation of a program in processor-1, a sandwich expression (not shown) causes the application *(f x)* to be transmitted to processor-2, where it is rewritten to *own x* (figure 16a).

$f\ x \rightarrow \ own\ x$

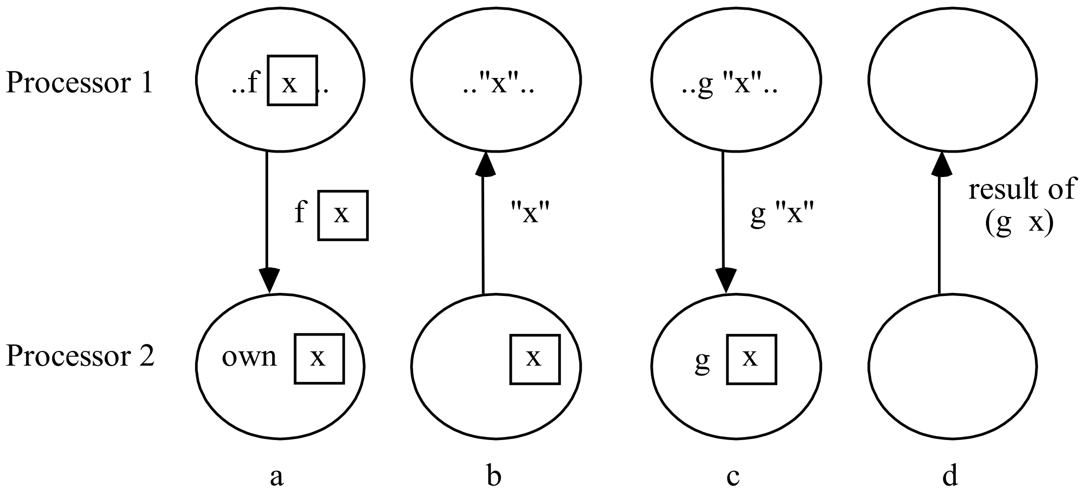


Figure 16: Retention of *x* using the *own* annotation

Reduction of the *own*-application in processor-2 results in the retention of the graph *x*, whereas a virtual value *"x"* is returned to processor-1 (figure 16b). This virtual value can not be considered as a regular pointer to *x*. Dereferencing *"x"* in processor 1 results in a fatal error. A virtual value may only be used in another sandwich expression. Suppose that an application *(g "x")* is part of a subsequent sandwich expression in processor-1 (again not shown) and as a consequence has to be transmitted to a remote processor. The application *(g "x")* will then be sent to the same processor as where *x* is retained. Upon arrival in processor-2, the virtual value *"x"* is replaced by the retained *x* (figure 16c). If no further applications of *own* occur the result of *(g x)* is normally returned to processor-1 (figure 16d).

Considering the definition of *G* in *(SW3)*, it seems impossible to retain the state information s_1 and s_2 by the use of the *own*-annotation. The state has to be returned to *G* on each recursion to compute $(g_1\ s_1)$ and $(g_2\ s_2)$. However, in many applications of type *(SW3)* these computations only yield small results compared to the size of their input arguments s_1 and s_2. Instead of returning the states s_1 and s_2, it seems more appropriate to compute $(g_1\ s_1)$ at the processor retaining s_1, and to compute $(g_2\ s_2)$ at the processor retaining s_2. Then, only the relatively small results of these applications have to be returned.

The following transformation of *(SW3)*, which we call the *own*-transformation, takes care of computing $(g_1\ s_1)$ and $(g_2\ s_2)$ in the appropriate processor:

```
G  ((s1 . a) . (s2 . b))                                        (SW4)
        →      a . G (Sandwich Cons (f1' s1 b) (f2' s2 a))
f1' s x →      (Own s1') . g1 s1'
                      s1': f1 s x
f2' s x →      (Own s2') . g2 s2'
                      s2': f2 s x
```

The state tuple of *(SW3)* has been extended in *(SW4)* with the result of computing $(g_1\ s_1)$ and $(g_2\ s_2)$. These computations are now part of the functions f_1' and f_2'. The latter two functions still compute the next state, using the old f_1 and f_2, but the state is retained in the remote processor by the annotation *Own*. Retention of the states is possible, because s_1 and s_2 are indeed nowhere used in the definition of *G* in *(SW4)*. They are merely passed as arguments to the applications of f_1' and f_2'. Thus if the implementation of *own* returns a virtual value, this value will never be dereferenced in *G*.

6 Using communication lifting

The communication lifting method for synchronous processes is sufficiently general to be implemented as an automatic development tool. Such a tool could take the definition of a synchronous process network and transform it into a single synchronous process. The sandwich transformations, however, cannot be easily automated, as they require knowledge of the grain-size of computations.

To demonstrate the proposed transformations as a programming method for the manual construction of parallel programs, we will apply them to the tidal model introduced in section 4.1 and to a logic-level simulation of digital hardware.

6.1 Transformation of the tidal model

To present a more complex example of communication lifting we will transform the parallel tidal model introduced in section 4.1, to obtain a mapping on the job-based reduction model. Figure 17 illustrates the communication structure (*GR*) of the non-simplified version of the program and introduces short unique names for the individual processes to obtain a reasonably compact notation while applying the rules T1-T6.

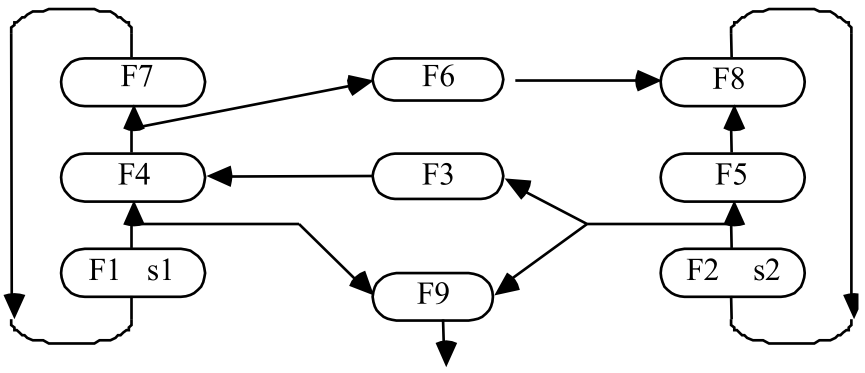


Figure 17: The structure of the tidal model

The definitions in *FN* and the graph *GR* associated with figure 17 are presented in figure 18:

FN::	$F_1\ s_1\ (x_1 . xr_1)$	$\rightarrow$	$s_1 . F_1\ x_1\ xr_1$
	$F_2\ s_2\ (x_2 . xr_2)$	$\rightarrow$	$s_2 . F_2\ x_2\ xr_2$
	$F_3\ (x_3 . xr_3)$	$\rightarrow$	$(g_3\ x_3) . F_3\ xr_3$
	$F_4\ (x_4 . xr_4)\ (y_4 . yr_4)$	$\rightarrow$	$(g_4\ x_4\ y_4) . F_4\ xr_4\ yr_4$
	$F_5\ (x_5 . xr_5)$	$\rightarrow$	$(g_5\ x_5) . F_5\ xr_5$
	$F_6\ (x_6 . xr_6)$	$\rightarrow$	$(g_6\ x_6) . F_6\ xr_6$
	$F_7\ (x_7 . xr_7)$	$\rightarrow$	$(g_7\ x_7) . F_7\ xr_7$
	$F_8\ (x_8 . xr_8)\ (y_8 . yr_8)$	$\rightarrow$	$(g_8\ x_8\ y_8) . F_8\ xr_8\ yr_8$
	$F_9\ (x_9 . xr_9)\ (y_9 . yr_9)$	$\rightarrow$	$(g_9\ x_9\ y_9) . F_9\ xr_9\ yr_9$
	$g_9\ x_9\ y_9$	$\rightarrow$	$x_9 . y_9$

GR::	a_1: F_1 mleft a_7	a_4: $F_4\ a_1\ a_3$	a_7: $F_7\ a_4$
	a_2: F_2 mright a_8	a_5: $F_5\ a_2$	a_8: $F_8\ a_5\ a_6$
	a_3: $F_3\ a_2$	a_6: $F_6\ a_4$	a_9: $F_9\ a_1\ a_2$

Figure 18: FN and GR of the tidal model

The rules F_3 to F_8 specify the update calculations that are applied to the physical quantities. A precise description of g_3 to g_8 can be found in [VRE87]. The functions receive and generate streams of matrices, without retaining any state information. The only two functions that contain a state are F_1 and F_2. They are *Cons*-processes as discussed in section 5.1. In *GR* the functions F_1 and F_2 are applied to their initial matrices: *mleft* respectively *mright*. Process F_9 merges the output of the left-hand side with that of the right-hand side into a single output stream. The node a_9 has to be considered as the output stream of *GR* (i.e. $a_{out} = a_9$).

We will first lift the processes of figure 18 into one single synchronous process. Next a transformation of type *(SW3)* yields a parallel program consisting of two coarse grain jobs. Application of the communication lifting steps T1-T6 proceeds as follows:

T1: The communication matrix corresponding to *GR* in figure 18 contains the following non-empty elements:

$C_{14} = \{x_4\}$, $C_{19} = \{x_9\}$, $C_{23} = \{x_3\}$, $C_{25} = \{x_5\}$, $C_{29} = \{y_9\}$, $C_{34} = \{y_4\}$,
$C_{46} = \{x_6\}$, $C_{47} = \{x_7\}$, $C_{58} = \{x_8\}$, $C_{68} = \{y_8\}$, $C_{71} = \{x_1\}$, $C_{82} = \{x_2\}$.

T2: From figure 18 it follows that $n = 2$, $(f_1\ s_1\ x_{11}\ x_{12}\ ...) = x_1$, $(f_2\ s_2\ x_{21}\ x_{22}\ ...) = x_2$, and we have assumed that$F_{out} = F_9$. Therefore *(T2)* yields the following replacements:

G [$(s_1 . s_2)$ / State]
G [$(x_1 . x_2)$ / New-state]
G [$(g_9\ x_9\ y_9)$ / Output]

T3: The contents of communication matrix *C* specifies the following replacements that have to be applied repeatedly to the expressions for *Output* and *New-state*:

x_1	→	$g_7\ x_7$	x_2	→	$g_8\ x_8\ y_8$	x_3	→	s_2
x_4	→	s_1	x_5	→	s_2	x_6	→	$g_4\ x_4$
y_4								
x_7	→	$g_4\ x_4\ y_4$	x_8	→	$g_5\ x_5$	x_9	→	s_1
y_4	→	$g_3\ x_3$	y_8	→	$g_6\ x_6$	y_9	→	s_2

During the replacements in the expression for *New-state* a common sub-expression arises and shared using label *a:*

New-state	→	$(x_1 . x_2)$
	→	$(g_7\ x_7) . (g_8\ x_8\ y_8)$
	→	$(g_7\ a) . (g_8\ (g_5\ s_2)\ (g_6\ a))$
a: $(g_4\ x_4\ y_4)$	→	a: $(g_4\ s_1\ (g_3\ x_3))$
	→	a: $(g_4\ s_1\ (g_3\ s_2))$

In the last step of rewriting *Output* the definition of g_9 is used:

```
Output              →    (g9 x9 y9)
                    →    (g9 s1 s2)
                    →    (s1 . s2)
```

T4: Because all variables x_{ik} of the processes F_i are contained in matrix C, the meta-variables *Input-streams* and *Stream-tails* are replaced by the empty list of arguments, resulting in the following single rewrite rule for the tidal model:

```
G (s1 . s2)     → (s1 . s2) . G ((g7 a) . (g8 (g5 s2) (g6 a)))
                  a: (g4 s1 (g3 s2))
```

T5 and T6: Replacing the formal arguments s_1 and s_2 of G by their corresponding actual arguments *mleft* and *mright* in *GR* yields the single application node that replaces *GR*:

```
a9: G mleft mright
```

The application at a_9 specifies a stream of matrix pairs that represent the state of the tidal model at successive time steps. A main expression, steering the program, may select a subset of these pairs to be printed.

From the tidal model it appears that the functions g_4, g_5, g_7 and g_8 perform a lot of computations. A transformation of type *(SW3)* can be applied to G to generate parallel jobs for the applications of g_7 and g_8:

```
G (s1 . s2)     → (s1 . s2) . (G c)
                  c: Sandwich Cons (g7 a) (g8 (g5 s2) (g6 a))
                  a: (g4 s1 (g3 s2))
```

The *Sandwich* strategy will first sequentially reduce the arguments *a*, *(g_5 s_2)* and *(g_6 a)* to normal form, before dispatching the jobs *(g_7 ...)* and *(g_8 ...)*. In this sequential evaluation we observe again the presence of two independent coarse grains of computation: the application of g_4 and g_5. The possibility to reduce these two applications in parallel too, leads to the final parallel version of the tidal model:

```
G (s1 . s2)     → (s1 . s2) . (G c)
                  a: Head b
                  b: Sandwich Cons (g4 s1 (g3 s2)) (g5 s2)
                  c: Sandwich Cons (g7 a) (g8 (Tail b) (g6 a))
```

Still a transformation of type *(SW4)* has to be applied in order to profit from the retention of the large state matrices (s_1 and s_2) in their respective remote processors. This transformation has been elaborated in [VRE88] where the version of G derived here, is considered as a given source program to be transformed. A performance figure of the transformed parallel tidal model on our experimental machine is presented in [HAR88].

6.2 Transformation of a digital hardware simulation

The simplified tidal model of figure 4 consists of two coarse grain synchronous processes without state information. We now show the transformation of a program where all processes contain state information and represent fine grain calculations. This program simulates digital hardware, using synchronous process definitions for the elementary components and "glueing" those components together with streams. Starting with the specification of a nand-gate as a synchronous process, a two-stage edge-triggered flipflop (D-type) is constructed. Assuming the flipflop has a sufficiently coarse grain size, we transform the stream definition of a shift-register based on these flipflops into a job-based parallel version.

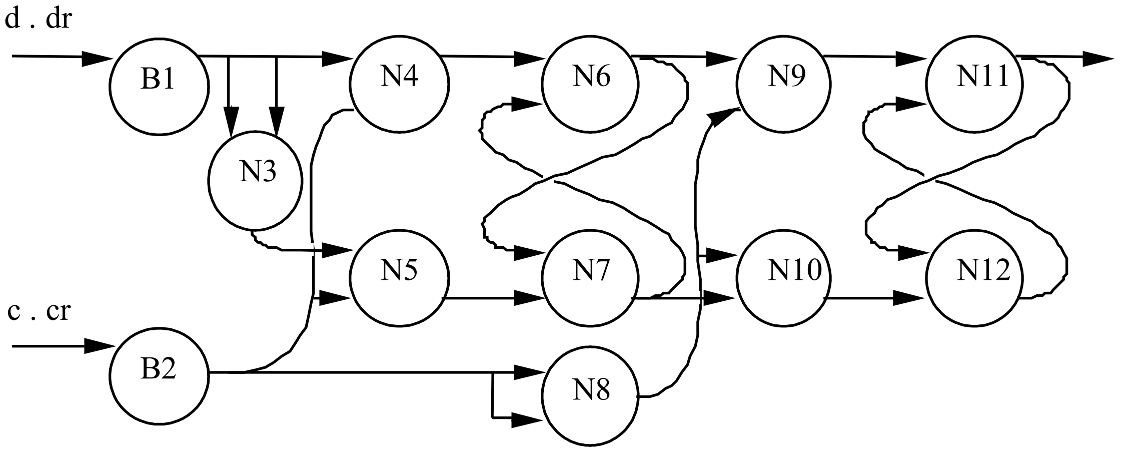


Figure 19: A stream definition of a D-type flipflop

The definition of a 2-input nand-gate with a unit time delay as a synchronous process, is as follows:

```
N s (x . xr) (y . yr)   → s . N (f x y) xr yr
                f x y   → Not (And x y)
```

The state of the nand-gate implements the delay of such a digital circuit. The definition of *N* shows that the application of the nand-function on the input elements *x* and *y* is first transferred into the internal state, to be delivered as an output element on the next recursive rewrite of *N*. A hardware description of a D-type flipflop, with two buffers B_1, B_2 and ten nand-gates numbered N_3 to N_{12}, can be translated into a stream-based definition of synchronous processes (see figure 19).

The input stream elements are called *d* and *c*, according to the meaning of these streams as data and clock. The two buffers B_1 and B_2 are both the identity process on streams with one unit time-delay. They have no other function than to provide the power to drive multiple internal circuits without presenting a multiple load to the outside world.

Figure 20 shows the function definitions (*FN*) and the interconnection graph (*GR*) of figure 19. Subscripts are used to obtain unique names for all variables. Note that in contrast to the previous examples some of the actual arguments in *GR* are constants. They specify the initial states of the circuits when power is switched on. These initial states can be arbitrary chosen from {0,1} denoting "false", respectively "true". (however, the program only works correctly if $s_7 = Not(s_6)$ and $s_{12} = Not(s_{11})$ because the flipflop has no external "clear" input). The illustration of figure 19 suggests that node a_{11} represents the output stream of *GR*. The actual input streams to *GR* are called "data" and "clock".

$$
\begin{array}{llcl}
\text{FN::} & B_1\ s_1\ (x_1 . xr_1) & \rightarrow & s_1 . B_1\ x_1\ xr_1 \\
 & B_2\ s_2\ (x_2 . xr_2) & \rightarrow & s_2 . B_2\ x_2\ xr_2 \\
 & N_i\ s_i\ (x_i . xr_i)\ (y_i . yr_i) & \rightarrow & s_i . N_i\ (f\ x_i\ y_i)\ xr_i\ yr_i\ \ i = 3,4, .. 12
\end{array}
$$

GR::	a_1: B_1 0 data	a_5: N_5 0 a_3 a_2	a_9: N_9 0 a_6 a_8
	a_2: B_2 0 clock	a_6: N_6 0 a_4 a_7	a_{10}: N_{10} 0 a_8 a_7
	a_3: N_3 0 a_1 a_1	a_7: N_7 1 a_6 a_5	a_{11}: N_{11} 0 a_9 a_{12}
	a_4: N_4 0 a_1 a_2	a_8: N_8 0 a_2 a_2	a_{12}: N_{12} 1 a_{11} a_{10}

Figure 20: FN and GR of the D-type flipflop

Transformation of the twelve gates into one process *Fp* according to the communication lifting rules T1-T6 runs as follows:

T1: The communication matrix corresponding to figure 20 contains the following non-empty sets:

$C_{1,3} = \{x_3, y_3\}$	$C_{3,5} = \{x_5\}$	$C_{7,6} = \{y_6\}$	$C_{10,12} = \{y_{12}\}$
$C_{1,4} = \{x_4\}$	$C_{4,6} = \{x_6\}$	$C_{7,10} = \{y_{10}\}$	$C_{11,12} = \{x_{12}\}$
$C_{2,4} = \{y_4\}$	$C_{5,7} = \{y_7\}$	$C_{8,9} = \{y_9\}$	$C_{12,11} = \{y_{11}\}$
$C_{2,5} = \{y_5\}$	$C_{6,7} = \{x_7\}$	$C_{8,10} = \{x_{10}\}$	
$C_{2,8} = \{x_8, y_8\}$	$C_{6,9} = \{x_9\}$	$C_{9,11} = \{x_{11}\}$	

T2: From figure 20 it follows that $n = 12$, which yields the following replacement for the meta-variable *State*:

$$Fp\ [\ (s_1 . s_2 . .... s_{12})\ /\ \text{State}\ ]$$

Composing all state expressions of B_1, B_2, N_3, - N_{12} into a tuple yields:

$$Fp\ [\ x_1 . x_2 . (f\ x_3\ y_3) . (f\ x_4\ y_4) . .... (f\ x_{12}\ y_{12})\ /\ \text{New-state}\ ]$$

We have assumed that $F_{out} = N_{11}$. In figure 20 the corresponding output generating expression is just the variable s_{11}, thus:

$$Fp\ [\ s_{11}\ /\ \text{Output}\ ]$$

T3: The communication matrix *C* yields the following replacements to be applied to the expressions for *Output* and *New-state*:

New-state [	s_1 / x_3	s_1 / y_3	s_1 / x_4	s_2 / y_4	s_2 / y_5	s_2 / x_8
	s_2 / y_8	s_3 / x_5	s_4 / x_6	s_5 / y_7	s_6 / x_7	s_6 / x_9
	s_7 / y_6	s_7 / y_{10}	s_8 / y_9	s_8 / x_{10}	s_9 / x_{11}	s_{10} / y_{12}
	s_{11} / x_{12}	s_{12} / y_{11}]				

which results in the final replacement for *New-state*:

```
Fp [ x1 . x2 . (f s1 s1) . (f s1 s2) . (f s3 s2) . (f s4 s7) . (f s6 s5) . (f s2 s2) .
         (f s6 s8) . (f s8 s7) . (f s9 s12) . (f s11 s10) / New-state ]
```

T4: The stream variables of *FN* not contained in the communication matrix are: x_1 and x_2. The argument lists of *Fp* thus become:

```
Fp [ (x1 . xr1) (x2 . xr2) / Input-streams ]
Fp [ xr1 xr2 / Stream-tails ]
```

Replacing the meta-variables in the skeleton of *Fp* according to the results obtained in T1-T4, yields a synchronous process for the flip-flop of figure 20:

```
Fp (s1 . s2 . .... s12) (x1 . xr1) (x2 . xr2)
            → s11 . Fp a xr1 xr2
               a: x1 . x2 . (f s1 s1) . (f s1 s2) . (f s3 s2) . (f s4 s7) . (f s6 s5) .
                  (f s2 s2) . (f s6 s8) . (f s8 s7) . (f s9 s12) . (f s11 s10)
```

Figure 21: The lifted process for a D-type fliplop

T5, T6: To derive the single application that is going to replace *GR* we have to transform the left hand side of the definition of *Fp* of figure 21. The formal state variables s_i and the formal stream arguments *(x_1 . xr_1)*, *(x_2 . xr_2)* are replaced by the corresponding actual arguments in *GR*, yielding:

```
a11: Fp (0 . 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 1) data clock
```

The preceding communication lifting of several nand-gates into one process definition of a flipflop is an example of how to enlarge the grain size of many fine grain processes into one coarse grain synchronous process. The coarse grain process can now be used in other groupings, where parallel execution is required. We will illustrate this by designing a parallel shift register based on the previously derived definition of *Fp* and performing another communication lifting on a group of flipflops.

To use the flipflop in the construction of a shift register, we introduce the name *ff* for the state-transforming function of *Fp* and the name *gf* for the output-generating function of *Fp*:

```
ff (s1 . s2 . .... s12) x1 x2   →   x1 . x2 . (f s1 s1) . (f s1 s2) . (f s3 s2) .
                                    (f s4 s7) . (f s6 s5) . (f s2 s2) . (f s6 s8) .
                                    (f s8 s7) . (f s9 s12) . (f s11 s10)
```

gf $(s_1 . s_2 . .... s_{12})$ → s_{11}

The stream based definition of for example a two bit shift register is obtained by concatenating two flipflops, sharing the same buffered clock and the same initial state:

```
FN::  B1 s1 (x1 . xr1)                →   s1 . B x1 xr1
      Fp2 s2 (x2 . xr2) (y2 . yr2)    →   (gf s2) . Fp2 (ff s2 x2 y2) xr2 yr
      Fp3 s3 (x3 . xr3) (y3 . yr3)    →   (gf s3) . Fp3 (ff s3 x3 y3) xr3 yr3

GR::  a1: B1 0 clock
      a2: Fp2 state data a1
      a3: Fp3 state a2 a1
```

The output stream of *GR* is produced by node *a1* and the unconnected inputs are *state*, *data* and *clock*. The communication lifting rules T1-T6 can be applied again to result in a lifted process *Sr* that replaces *FN*:

```
Sr (s1 . s2 . s3) (x1 . xr1) (x2 . xr2)
      → (gf s3) . Sr (x1 . (ff s2 x2 s1) . (ff s3 (gf s2) s1)) xr1 xr2
```

and an application that replaces *GR*:

```
a1: Sr (0 . state . state) clock data
```

Subsequently *Sr* can be transformed into a job-based version following *(SW3)*:

```
Sr (s1 . s2 . s3) (x1 . xr1) (x2 . xr2)
      → (gf s3) . Sr (x1 . a) xr1 xr2
           a: Sandwich Cons (ff s2 x2 s1) (ff s3 (gf s2) s1)
```

To retain the considerable amount of state information of a flip-flop in the processor to which the flip-flop will be allocated, a transformation of type *(SW4)* may be applied:

```
Sr (s1 . (s2 . gfs2) . (s3 . gfs3)) (x1 . xr1) (x2 . xr2)
      → gfs3 . Sr (x1 . a) xr1 xr2
           a: Sandwich Cons (ff' s2 x2 s1) (ff' s3 gfs2 s1)

ff' s x y      → (Own a) . (gf a)
                   a: ff s x y
```

The sandwich annotation distributes the computation of the state-transforming functions *ff'* of the flip-flops. The results of these computations consist of a new state and a new output-element for each flip-flop. The new state is retained in the remote processors, whereas the output-elements are actually returned to the shift-register process.

7 Conclusion

The job-based model for parallel reduction mediates between pure graph reduction and string reduction. The model allows efficient parallel reduction of certain application programs on architectures without shared memory. In our approach parallel jobs have to be annotated by the programmer. However, application programs written as process networks, containing streams and cyclic structures, do not fit directly in this model. Three transformations are presented, called *communication lifting*, *sandwich*- and *own-transformation*. Together they allow a subset of applications written as synchronous process networks to be mapped onto the job-model

Streams and cyclic structures are frequently present in functional programs that model process networks. Using a linear notation for graph rewriting, the reduction of such a program demonstrates that cyclic structures disappear from the graph in an early stage. An equivalent non-cyclic program is shown to yield the same computational structure.

Using the example as a guide-line, a general method is presented that allows the transformation of a (cyclic) network of processes into *one* non-cyclic process. The method, called communication lifting, is applicable to a network of processes that behave according to a model, in which communication between processes occurs synchronously. For application programs written according to this model, a set of formal transformation rules is presented describing communication lifting.

For a synchronous process two other transformations are informally presented, the *sandwich*- and *own-transformation*. These allow an efficient mapping of the process onto our job-based parallel reduction model. The job model has to be extended by an extra annotation that causes the retention of graphs in remote processors.

For two application programs it is shown how to construct and annotate coarse grain parallel jobs, using the presented transformations. One application program is a tidal model of the North Sea, consisting of coarse grain communicating processes. The other application shows how to construct a simple parallel simulation of digital hardware, from fine grain processes.

8 Acknowledgements

We gratefully acknowledge the support of the Dutch parallel reduction machine project team. Pieter Hartel, Marcel Beemster and Henk Muller made valuable comments on draft versions of the paper.

9 References

[BAR87a] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, M.R. Sleep, "Term graph rewriting", in Conf. on Parallel Architectures and Languages Europe (PARLE), part II, Eindhoven, the Netherlands, LNCS vol 259, pp. 141-158, June 1987.

[BAR87b] H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, M.R. Sleep, "Towards an intermediate language based on graph rewriting", in Conf. on Parallel Architectures and Languages Europe (PARLE), part II, Eindhoven, the Netherlands, LNCS vol 259, pp. 159-175, June 1987.

[BAR87c] H.P. Barendregt, M.C.J.D. van Eekelen, M.J. Plasmeijer, P.H. Hartel, L.O. Hertzberger, W.G. Vree, "The Dutch parallel reduction machine project", Future generations computer systems, Vol 3, No 4, pp 261-270, Dec 1987.

[BRU87] T.H. Brus, M.C.J.D. van Eekelen, M.O. van Leer, M.J. Plasmeijer, "Clean: a language for functional graph rewriting", in Third conf. on functional programming languages and computer architecture, Portland , Oregon, USA, LNCS 274, pp. 364-384, September 1987.

[HAR87] P.H. Hartel, A.H. Veen, "Statistics on graph reduction of SASL programs, Software practice and experience, Vol. 18, No. 3, March 1988, pp 239-253

[HAR88] P.H. Hartel, W.G. Vree, Parallel graph reduction for divide and conquer applications - part II, PRM project internal report D-20, Dept. of Comp. Sys., Univ. of Amsterdam, Dec. 1988.

[HER89] L.O. Hertzberger, W.G. Vree, "A Coarse Grain Parallel Architecture for Declarative Languages", in Conf. on Parallel Architectures and Languages Europe (PARLE), Eindhoven, the Netherlands, LNCS vol. 365, pp 269-285, June 1989

[HUD85] P. Hudak, B. Goldberg, "Distributed execution of functional programs using serial combinators", IEEE Transactions on computers, Vol. C-34, No. 10, pp 881-891, Oct. 1985.

[JOH84] T. Johnsson, "Efficient compilation of lazy evaluation", Proc. of the ACM Sigplan '84, Sigplan Notices, Vol. 19, No 6, June 1984, pp 58-69.

[KAH74] G. Kahn, "The semantics of a simple language for parallel programming", in Information Processing 74, North-Holland Publishing Company, pp 471-475

[KEL89] P. Kelly, "Functional programming for loosely-coupled |multiprocessors", in series 'reserach monographs in parallel and distrbuted computing', Pitman & MIT Press, April 1989

[MCB87] D.L. McBurney, M.R. Sleep, "Transputer-based experiments with the ZAPP architecture", in Conf. on parallel architectures and languages Europe (PARLE), part I, Eindhoven, the Netherlands, LNCS 259, pp. 242-259, June 1987.

[PEY87a] S.L. Peyton-Jones, "The implementation of functional programming languages", Prentice Hall, Englewood Cliffs, New Jersey, 1987

[PEY87b] S.L. Peyton-Jones, C. Clack, J. Salkild, M. Hardie, "GRIP-a high performance architecture for parallel graph reduction", in Third conf. on functional programming languages and computer architecture, Portland , Oregon, USA, LNCS 274, pp. 98-112, September 1987.

[VRE87] W.G. Vree, "The grain size of parallel computations in a functional program", in Conf. on parallel processing and applications, l'Aquila, Italy, September 1987, Elsevier Science Publishing, pp 363-370.

[VRE88] W.G. Vree, P.H. Hartel, "Parallel graph reduction for divide and conquer applications - part I", PRM project internal report D-15, Dept. of Comp. Sys., Univ. of Amsterdam, December 1988.

[WAT87] P. Watson, I. Watson, "Evaluating functional program on the FLAGSHIP machine", in Third conference on functional programming languages and computer architecture, Portland , Oregon, USA, LNCS 274, pp. 80-97, September 1987.

[WRA86] S.C. Wray, "Implementing and programming techniques for functional languages", PhD. thesis, Tech. Rep. 92, Univ. of Cambridge, Jan 1986.

Appendix: Summary of communication lifting

A.1 Models of a synchronous process

$F\ s\ (x_1 \,.\, xr_1) \ldots (x_n \,.\, xr_n) \rightarrow (g\ s\ x_1 \ldots x_n) \,.\, F\ (f\ s\ x_1 \ldots x_n)\ xr_1 \ldots xr_n$ (S1)

where	s	is the state of the process F
	$(x_i \,.\, xr_i)$	are input streams to F
	g	is a function that computes the next output element of F
	f	is a function that computes the next state of F

$F\ (x_1 \,.\, xr_1) \ldots (x_n \,.\, xr_n) \rightarrow (g\ x_1 \ldots x_n) \,.\, F\ xr_1 \ldots xr_n$ (S2)

A.2 Application requirements

The application to be transformed is a rule set *FN* and a graph *GR* with the following restrictions:

FN:: $F_i\ s_i\ (x_{i1} \,.\, xr_{i1})\ (x_{i2} \,.\, xr_{i2}) \ldots$ (S3)

$\rightarrow (g_i\ s_i\ x_{i1}\ x_{i2} \ldots) \,.\, F_i\ (f_i\ s_i\ x_{i1}\ x_{i2} \ldots)\ xr_{i1}\ xr_{i2} \ldots$ (i = 1.. n)

$F_i\ (x_{i1} \,.\, xr_{i1})\ (x_{i2} \,.\, xr_{i2}) \ldots$

$\rightarrow (g_i\ x_{i1}\ x_{i2} \ldots) \,.\, F_i\ xr_{i1}\ xr_{i2} \ldots$ (i = n+1.. m)

GR:: $a_i : F_i\ t_i\ b_{i1}\ b_{i2} \ldots$ (i = 1.. n) (S4)

$a_i : F_i\ b_{i1}\ b_{i2} \ldots$ (i = n+1.. m)

$\forall i \in 1..m \mid (a_i : F_i \ldots) \in GR \quad F_i \in FN$ (S5)

$\exists!\ out \in 1..m \mid a_{out} : F_{out}\ s_{out}\ b_{out,1}\ b_{out,2} \ldots$ is the output of GR (S6)

A.3 The description of communication lifting

Model rule:

New-FN:: G State Input-streams → Output . G New-state Stream-tails

Transformation of *FN* and *GR* with the model rule in six steps yielding *NEW-FN* and *NEW-GR*:

$\forall i, j \in 1..\, m$ (T1)

$\forall k \in 1\,..$ the number of stream arguments of F_j

iff GR contains two nodes a_i , a_j such that:

$\mathbf{a_i}$: F_i t_i b_{i1} b_{i2} ...

a_j: F_j t_j b_{j1} ... $b_{j,k-1}$ $\mathbf{a_i}$ $b_{j,k+1}$ ....

and the definition of F_j in FN is:

F_j s_j $(x_{j1} . xr_{j1})$ ... $(\mathbf{x_{jk}} . xr_{jk})$ ... $\rightarrow$ g_j ... . F_j ...

then $\mathbf{x_{jk}} \in C_{ij}$

G [$(s_1 . s_2 . \ldots . s_n)$ / State] (T2)

G [$(f_1\ s_1\ x_{11}\ x_{12}\ \ldots) . (f_2\ s_2\ x_{21}\ x_{22}\ \ldots) . \ldots . (f_n\ s_n\ x_{n1}\ x_{n2}\ \ldots)$ / New-state]

G [$g_{out}\ s_{out}\ x_{out,1}\ x_{out,2}\ \ldots$ / Output]

$\forall i, j \in 1..\, m$ *and* $\forall k \in 1\,..$ the number of stream arguments of F_j (T3)

if $x_{jk} \in C_{ij}$

then G [$(g_i\ s_i\ x_{i1}\ x_{i2}\ \ldots)$ / x_{jk}]

G [(argument-list of all $(x_{jk} . xr_{jk})$ such that $x_{jk} \notin C_{ij}$) / Input-streams] (T4)

G [(argument-list of all xr_{jk} such that $x_{jk} \notin C_{ij}$) / Stream-tails]

where $i, j \in 1..\, m$ and $k \in 1..$ number of stream arguments of F_j

$\forall k \in 1..n$: (T5)

$$\left.\begin{array}{l} New\text{-}\ FN :: G(s_1 \ldots \mathbf{s_k} \ldots s_n) \ldots \rightarrow \ \ldots \\ GR :: a_{\mathbf{k}} : F_k \mathbf{t_k} b_{k1} b_{k2} \ldots \end{array}\right\} \Rightarrow \text{New-GR:: } a_{out}\text{: G } (s_1 \ldots \mathbf{t_k} \ldots s_n) \ldots$$

($\forall$ i, j $\in$ 1..m and $\forall$ k $\in$ 1..number of stream arguments of F_j) | $x_{jk} \notin C_{ij}$: (T6)

$$\left.\begin{array}{l} New\text{-}\ FN :: G(\ldots)\ldots(\mathbf{x_{jk}} . \mathbf{xr_{jk}}) \ldots \rightarrow \ \ldots \\ GR :: a_{\mathbf{j}} : F_j t_j b_{j1} \ldots \mathbf{b_{jk}} \ldots \end{array}\right\} \Rightarrow \text{New-GR:: } a_{out}\text{: G } (\ldots) \ldots \mathbf{b_{jk}} \ldots$$

A.4 Sandwich and own transformations

Model program:

G $(s_1 . s_2)$ $\rightarrow$ a . G $((f_1\ s_1\ b) . (f_2\ s_2\ a))$

a: g_1 s_1

b: g_2 s_2

Transformation examples SW1-SW4 of the model program:

G $(s_1 . s_2)$ $\rightarrow$ a . G (Sandwich Cons $(f_1\ s_1\ b)$ $(f_2\ s_2\ a)$) (SW1)

a: Head c

b: Tail c

c: Sandwich Cons $(g_1\ s_1)$ $(g_2\ s_2)$

$G\ (s_1 . s_2) \quad \rightarrow \quad a\ .\ G\ ((f_1\ s_1\ b) . (f_2\ s_2\ a))$ (SW2)

$a\text{: Head } c$

$b\text{: Tail } c$

$c\text{: Sandwich Cons } (g_1\ s_1)\ (g_2\ s_2)$

$G\ (s_1 . s_2) \quad \rightarrow \quad a\ .\ G\ (\text{Sandwich Cons } (f_1\ s_1\ b)\ (f_2\ s_2\ a))$ (SW3)

$a\text{: } g_1\ s_1$

$b\text{: } g_2\ s_2$

$G\ ((s_1 . a) . (s_2 . b))$ (SW4)

$\rightarrow \quad a\ .\ G\ (\text{Sandwich Cons } (f_1'\ s_1\ b)\ (f_2'\ s_2\ a))$

$f_1'\ s\ x \quad \rightarrow \quad (\text{Own } s_1') . g_1\ s_1'$

$s_1'\text{: } f_1\ s\ x$

$f_2'\ s\ x \quad \rightarrow \quad (\text{Own } s_2') . g_2\ s_2'$

$s_2'\text{: } f_2\ s\ x$

Chapter VIII

PARALLEL GRAPH REDUCTION FOR DIVIDE-AND-CONQUER APPLICATIONS
PART 2 - PROGRAM PERFORMANCE[1]

[1] PRM project internal report D-20, Dept. of Comp. Sys., Univ. of Amsterdam, December 1988

Parallel graph reduction for divide-and-conquer applications[†] Part II - program performance

Pieter H. Hartel

Willem G. Vree

Computer Systems Department, University of Amsterdam
Kruislaan 409, 1098 SJ Amsterdam

Abstract

An extensible machine architecture is devised to efficiently support a parallel reduction model of computation. The organisation of the machine is designed to match the behaviour of the application programs. A pilot implementation of the architecture is used to obtain an execution profile of the various applications. These profiles are used with a performance model to calculate optimal schedules of the applications. The resulting speedup figures give an upper bound for the performance gain that may be attained on a full implementation of the architecture. The most important result is that each application allows for a processor utilisation of over 50% to be attained on our parallel architecture.

Key words: local memory architecture multiple processor system optimal scheduling parallel graph reduction performance measurement

1. Introduction

With today's microprocessor technology it is possible to connect large numbers of powerful processors via a high speed communication network. Each processor may be equipped with a large store, to which it has high speed access. Storage modules can be equipped with few access ports. Arbitration logic makes shared access possible, with the same high speed, unless a storage cell is accessed from more than one port at exactly the same time. It is difficult to provide a large number of processors with high speed access to a common store. A globally shared component tends to reduce fault tolerance, extensibility and potential parallelism of a system. Considering this, we set out to develop a model of computation based on reduction,

† This work is supported by the Dutch Ministry of Science and Education, dienst Wetenschapsbeleid

that can be implemented efficiently on an architecture without a common store. In part I of this paper[1] it was shown, that based on this model of computation, interesting application programs, such as Wang's algorithm[2] to solve a sparse system of linear equations, can be transformed into functionally equivalent versions that benefit from parallel evaluation on such an architecture.

The model of computation is based on the concept of a job. This is a closed, needed redex that can be evaluated in parallel to other jobs at a cost that can be kept low for two reasons. Firstly, during the evaluation of a job, there is no need for communication since it is closed. Secondly, the communication costs incurred in setting up the job on a separate processor and returning its results can be kept low enough to make parallel evaluation beneficial. This is achieved by transforming programs without this property into functionally equivalent ones with this property. A possible disadvantage of this scheme is, that parallel evaluation of closed expressions makes it necessary to duplicate shared subexpressions. To avoid the duplication of work, such subexpressions must be in normal form. A function is available to normalise shared subexpressions before the generation of parallel jobs.

Jobs arise when a special function "sandwich" is encountered during the evaluation of an application. It gives the arguments of the function the status of a job and schedules their parallel evaluation. The application programmer has to ensure, that the requirements for jobs are indeed satisfied. Special precautions may have to be taken to balance communication and reduction cost. For instance the recursive subdivision of unsorted lists in the quick sort algorithm must be stopped when the lists become too small. A threshold mechanism achieves this form of dynamic grain size control. Applications that lend themselves well to be written as "sandwich" programs are divide-and-conquer algorithms.

In this part of our paper we describe the machine model in more detail and present performance figures with respect to the application programs and a pilot implementation of the architecture.

2. Machine model

The architecture of the parallel reduction machine that we use to support the sandwich strategy consists of a network of processing elements, each with a fair amount of local store. We do not make assumptions about the topology. Until now we have used a string of processing elements and experiments with a regular mesh structure are planned. The use of shared store as a communication device allows for some interesting optimisations to be implemented.

2.1. Storage

The storage space of a processing element is the set of storage cells that can be accessed by elementary operations, such as "dereference pointer" or "allocate cell". This is called local access. Although in general communication facilities are necessary to access the store of an arbitrary processing element (non-local access), the storage spaces of adjacent processing elements may partly overlap. Hence some transactions may bypass the communication facilities, because both parties have local access to the same store. When individual storage cells are addressed, non-local access is always much slower than local access. Most communication systems transfer large groups of elementary data items as a single packet to amortise the overhead incurred in setting up a transaction.

The classical message passing paradigm does not take advantage of overlapping storage. This is mainly due to the call by value semantics of the message passing primitives, which causes a message to be copied from source to destination. Yet another copy of the message has to be made if during transmission the destination storage area is still unknown. This unfortunate situation arises because data transfer is usually combined with process synchronisation and it may well occur that the recipient of the message is not yet ready to accept it. One solution is to delay the transmitter until the recipient is prepared to communicate, but this is unacceptable in those areas where insufficient parallelism is available to cover the waiting periods. Regular message passing causes at least two copies to be made of the transported message. Not even a single copy is necessary if both parties in communication have access to the same local store and synchronisation is separated from communication. The latter scheme is used in our proposal to transport jobs and results.

2.2. Processing

An alternative name for string reduction is tree reduction. This term blends well with the "job" structure that is generated by the sandwich strategy. The root of the tree is formed by the main job. Reduction of a sandwich expression causes new jobs to be created. The representation of a job "flows" along the edge that connects the job to its parent. On termination, a job communicates the result to its parent along the same edge but in opposite direction. Communication between two jobs is only possible, when they are parent and child. Consider as an example the job structure shown in figure (1) that arises during the execution of Wang's partitioning algorithm. The horizontal solid lines represent sequential calculations (measured in reduction steps). The vertical solid lines represent the size of the jobs (measured as a number of nodes) that are transmitted to be reduced in parallel. The computation starts off sequentially (185 steps) until the first two subjobs are created. One of them causes two new jobs to be started until we arrive at the situation where five jobs are evaluated in parallel for a relatively long time. In order not to clutter up the diagram the jobs and results are shown as separate trees. The flow of results is drawn as dashed lines that mirror the flow of jobs. In most applications that we have run it takes little time to merge the results. Wang's algorithm consists of two parallel phases and a sequential phase: after the first elimination phase a long sequential calculation is necessary (7411 steps) before the second elimination phase can be started.

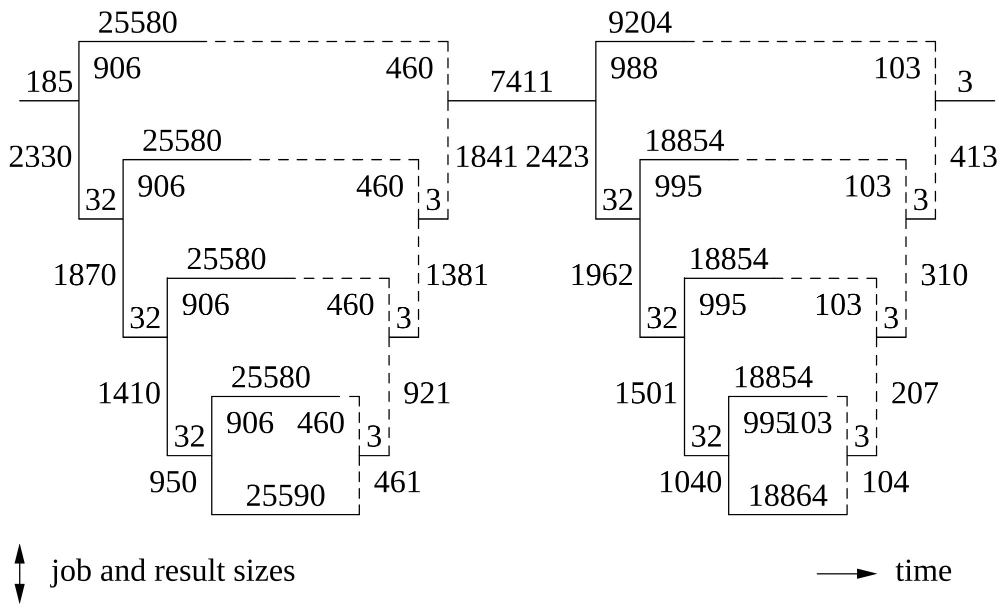


Figure 1 : The job and result trees generated by Wang's algorithm (not drawn to scale)

The processing elements in the parallel machine architecture must be arranged in such a way that a dynamically generated job-tree as described above can be mapped on the physical topology. Each individual processing element must be capable of supporting more than one reducer (process) and a reducer is involved with a single job until the job terminates. Within a processing element a form of local scheduling is necessary to allow for a reducer to wait for completion of the children of the job it is reducing. The processing element is then free to take up another assignment. By definition the normal form of a job is needed in some context, hence the local scheduling need not be concerned with preemption and rescheduling of active jobs.

If the number of jobs does not exceed the number of processing elements, each processing element could be allocated to a job (via a reducer). In that case the utilisation of resources is not optimal. Therefore the number of jobs should be larger than the number of processing elements. Indiscriminate allocation of jobs to reducers may not yield good results. For instance if all leaf nodes in the job-tree end up in a single processing element, the overall performance of the system will be worse than that of a sequential machine.

2.2.1. The Conductor

Control is necessary to spread the jobs over the available processing elements and to make sure that the storage requirements of the jobs do not exceed the machine capacity. Both activities require global information. To achieve this, we have decided to allocate this task to a dedicated processing element. We call this centralised scheduler the conductor to stress that it has complete control over the "orchestra" of reducers, but that once a reducer has been allocated a job, it enters a relatively long period of autonomy. To be responsive, the conductor must have a "direct" connection to each reducer. In large systems it will be necessary to implement the conductor in a distributed fashion. Each single conductor controls a section of the system, but by exchange of information between conductors, global control of the system is still effectuated. Our expectation is that this organisation does not introduce a bottleneck, since the purpose of creating jobs was to produce large grains of parallel computation. If the jobs are too small to sustain the extra cost incurred in centralised control, the tools that were developed to regulate the grain size were applied inappropriately.

The task of the conductor is to balance the load in an environment with resources that are scarce. In general there are many ways to distribute a number of jobs over a number of processing elements. Each possible distribution is called a schedule. Not all schedules are feasible, because the storage capacity of each processing element is limited. The schedules that would cause the capacity of one or more processing elements to overflow should be rejected. It is the purpose of the conductor to choose the shortest feasible schedule. A practical load distribution algorithm can not guarantee that a feasible schedule is chosen, because the maximum size of a job is not known in advance. It is therefore possible that deadlock will occur. However, such a situation can be detected immediately. In a system with background store the risk of deadlock will be lower, because the storage capacity of each processing element will be larger.

To allow for the conductor to make sensible decisions, the size of a job has to be included in a request for job allocation. In the applications that were developed in part I of this paper, this information is already present for dynamic grain size control, so it can be used at no extra cost. The load balancing algorithm of the conductor will base its allocation policy on the recorded history of the application program that is running. In our opinion the history should also include information about previous runs of the same application, which given that most applications are run more than once, should in principle be possible. The behaviour of an application is captured in a parameterised “profile”. For example the quick sort algorithm has a profile shown in figure (2).

step	action	expression	interpretation
1.	select pivot	p_1	constant time
2.	split list	$l \times p_2$	time proportional to the length of the list
3.	recursively sort sublists	$l_1 \times p_3$	
		$l_2 \times p_4$	times dependent on the lengths of sublists
4.	append pivot and sublists	$l_1 \times p_5$	time dependent on length of first sublist

Figure 2 : Execution profile of quick sort

Fed with this information, the conductor can make estimates of the execution times of both recursive invocations of quick sort at the time they are about to be scheduled (step 3). The parameters p_3 and p_4 are multiplied by the lengths of the sublists, which are calculated by the split phase for the purpose of dynamic grain size control. In a sense the conductor is allowed to look one “step” ahead in time, which gives it predictive power to schedule the next family of jobs.

We are still investigating general methods for the specification of execution profiles.[3] Our current results are based on exact profiles of the applications, which state the real execution times rather than the parameters from which execution times can be estimated. The performance results presented in this paper are calculated a posteriori, from the recorded execution profiles. The calculation of the optimum schedule (see section 4) is based on a heuristic, which uses advance knowledge that is restricted to one “step”, such that the results provide an upper bound on the performance gain on a full implementation of the system.

2.2.2. The reducer

A reducer performs the actual rewriting of an expression into a normal form. To avoid the complexity of dynamic process creation, all reducers are started when the system is started. Steps 1 and 2 (below) are performed ad infinitum, by each reducer. Step 3 is performed when a sandwich expression is encountered.

1) The reducer waits until a job arrives. The job will require many reduction steps before it reaches head normal form, since it represents a coarse parallel grain.
2) The normal form of a job must be returned to its creator. The creator of the job will find that the root of the original representation of the job has been overwritten by the result.
3) The evaluation of a sandwich expression may cause new jobs to be created, provided enough resources are available: a free reducer and sufficient storage for each job. The conductor process will be asked permission before the jobs may be created. A single transaction with the conductor is sufficient, since all potential jobs are available at the same time. The reducer has to wait until the conductor sends its reply, otherwise it could alter the jobs (while reducing) and this would make the size of a job an unreliable measure. Another reason is, that after all jobs have been taken up by other reducers, there can be hardly any work left, such that the reducer might as well be suspended until all jobs are complete. If the conductor refuses the request, evaluation proceeds in the normal lazy fashion.

2.2.3. Graph transport

In addition to the reducers, each processing element supports a graph transfer process. This process operates like an interrupt handler, in the sense that when a message is received to transport a graph, normal (reduction) processing is interrupted, and the transport is effectuated as a single indivisible action. On completion, control is returned to the interrupted reducer. Like a real interrupt handler, the graph transport process should not encounter delays, such as those resulting from synchronisation requirements between producer and consumer of graphs. The reason that such delays are impossible is because all parties in the transfer of jobs or results are inactive while the transfer is taking place. The consumer of a graph is inactive because it is a reducer that is waiting for either a result or a new job. In the case of a result transfer, the producer has just reached a normal form, hence it can no longer be active. It was shown earlier, that it is necessary for the producer of a job to be suspended until the result appears.

Since a graph that arrives at its destination requires heap space, interaction between graph transport and reduction (via storage allocation) deserves further attention. Large graphs are transported in a number of packets and each packet contains a number of nodes. Depending on the particular storage allocator that is used, in one request an area may be allocated that is large enough to store the entire graph, a packet or just a node. The smaller the allocation unit, the more likely it is, that graph transport will be slow. Unfortunately storage allocation and reclamation schemes that support varisized allocation are more expensive than those that only support fixed size allocations.[4, 5] Hence there is a tradeoff between data communication speed and sequential reduction speed.

The graph transport mechanism that we have opted for assumes, that a contiguous block of store, large enough to hold the entire graph is allocated before the first packet arrives at its destination. The reasons for this choice are twofold. Firstly, the algorithm is simple enough to be implemented directly in hardware. Secondly it may also serve to perform copying garbage collection. In this way impaired sequential reduction speed can be improved significantly.

2.2.3.1. Copying garbage collection and graph transportation

The conditions that are satisfied when a graph transport operation is started can be summarised as follows. The transmitting process is guaranteed not to alter the graph that forms the contents of the message in $from - space$, because the entire process of graph transportation is an indivisible action to the transmitting process. The storage area of the message at the receiver side in $to - space$ is known in advance. The area is also reserved, because the allocation has already been done, for instance by the conductor.

To make an efficient hardware implementation possible, the number of accesses to $from -$ and $to - space$ must be minimised, since accessing non local (off board) information incurs considerable protocol overhead. The following classification of accesses may serve to clarify the restrictions imposed by such efficiency considerations:

Reading nodes at arbitrary locations in $from - space$

> During the copying process, each reachable node must at least be read once. A shared node is read as many times as there exist pointers to that node.

Writing pointer fields at arbitrary locations in $from - space$ (marking)

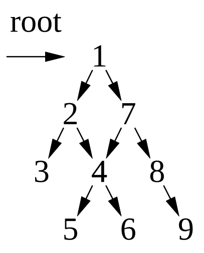
> Sharing requires the copying algorithm to mark the nodes that have already been processed. Marking may be performed by storing the forwarding address of a node in the original node in $from - space$.

Writing nodes in “stream mode” to $to - space$

> A node needs to be output once only, if the relevant information contained in the node

has been updated completely before it is output. This feature is a significant advantage, as it allows the nodes to be output as a continuous stream (into a pipeline), without the need for explicitly indicating the destination addresses of the nodes.

The compaction algorithm that we are using traverses the graph in pre-order. Entire nodes are read out and stored in a local stack. The address of the next node to be output is maintained in a local counter. It is incremented by the size of a node each time one is output to *to – space*. The stack contains the nodes, which form the leftmost path from the root to the current node. If the top of the stack contains a node that does not require any of its pointers to be updated any more, it is output to *to – space*. The stack is popped and the appropriate pointer in the new top node is replaced by the current value of the output counter. When a previously copied node is encountered, its forwarding address rather than the contents of the output counter is used. The algorithm is started, with a stack that contains a copy of the root and it terminates as soon as the stack has become empty. At the end an additional traversal of the graph in *from – space* is needed to reset the marks.



```
Initial stack configuration
     1 2 3
     1 2 4 5
     1 2 4 6
     1 2 4
     1 2     ——► stack growth
     1 74*
     1 7 8 9
     1 7 8
     1 7
     1
Final stack configuration
```

Figure 3 : (a) sample graph (b) successive stack configurations

The sample graph of figure (3-a) causes the stack configurations of figure (3-b) at the moments when a copy of a node is output to *to – space*. The cell marked with an asterisk is discovered to be a shared node.

2.2.3.2. Cyclic graphs

The graph compaction algorithm will fail to terminate if cycles are present in the graph. In functional programs, cycles can only be created by recursive functions. Within the body of a recursive function, the occurrence of the function name itself causes a cycle to be created in graph reduction. The number of functions however is determined by the compiler, and remains constant during execution. Pointers to functions within a graph can be implemented by constants, which represent the index in the table, where the function is stored. Hence these cycles will disappear.[5] The same reasoning also holds for mutually recursive functions.

The solution to the Hamming problem[6] uses a recursive data structure, which if properly implemented by a cyclic graph, results in a linear time algorithm. The cyclic data structure maintains a form of history, which can also be achieved by using explicit parameters to represent the history. The algorithm still runs in linear time, but no longer contains cycles. The same type of transformation can be used to eliminate cyclic data structures in a wide class of practical applications.[7] This transformation has been applied to one of our test programs (the tidal model). Compaction algorithms exist that can handle cyclic graphs properly, but these are less efficient. Either the graph must be traversed more than once, or the copied nodes are updated after they have been output. We propose to avoid cyclic graphs, even though certain computations will be performed less efficiently.

2.2.3.3. Performance analysis

An estimate is given of the expected performance of the graph compaction algorithm described above, both in case it is implemented in hardware and in software. The two implementations differ in several aspects:

Data transfer protocol cost

Some bus protocols allow for data to be transferred as a continuous stream, without intervening addresses. Both at the transmitting and the receiving side the address of the current datum, maintained in local registers, is incremented after each transfer. This allows the hardware implementation to have a much higher access rate to the *to – space* than a software implementation.

Instruction fetch and execution

The software implementation requires the CPU to fetch, decode and execute machine instructions. Our transfer algorithm was coded in 32 Motorola MC68010 machine instructions (78 bytes), of which on the average 90% are executed per node. These could be kept in the MC68020 on-chip instruction cache. In spite of its ability to overlap instruction decoding and execution, the MC68020 still requires time to execute some instructions (e.g. branches) that can not be overlapped with data transfers.

Hardware parallelism

Many operations that must be performed in sequential order by a general purpose processor, can be performed in parallel by a special purpose processor such as a graph compaction module. For example, the algorithm has been designed such, that once a node is ready, the original may be marked while the copy is being output to another store. Such an optimisation can only be achieved with hardware.

Arbitration protocol cost

The share of protocol cost in accessing the bus is not negligible. The CPU has insufficient means to optimise the usage of the bus, since the bus protocol circuitry enforces the use of a standard protocol. In contrast, the hardware implementation needs to acquire mastery over the destination bus once and may continue to use the bus as efficiently as possible.

The performance of a software implementation (on a MC68010) was found to exceed 10.000 nodes per second. A preliminary study has shown, that the hardware implementation can be up to two orders of magnitude faster.

The graph compaction algorithm has the disadvantage, that it requires a local stack, which on the average requires $\sqrt{n}$ cells for a graph with n nodes.[5] A stack of for instance ten thousand nodes with 2×32 bits per node does not pose unsurmountable problems. Because stack overflow can not be prevented nor ignored,[4] special precautions must be taken to deal with stack overflow properly.[8]

2.3. Cooperation of functional units

Having exposed the functionality of the components in the architecture, we can now show with an example how they cooperate. Figure (4) represents a configuration with three processing elements dedicated to reduction and the conductor. Graphs reside in overlapping stores. The life cycle of a single job is traced by describing, in chronological order, the messages that travel the system.

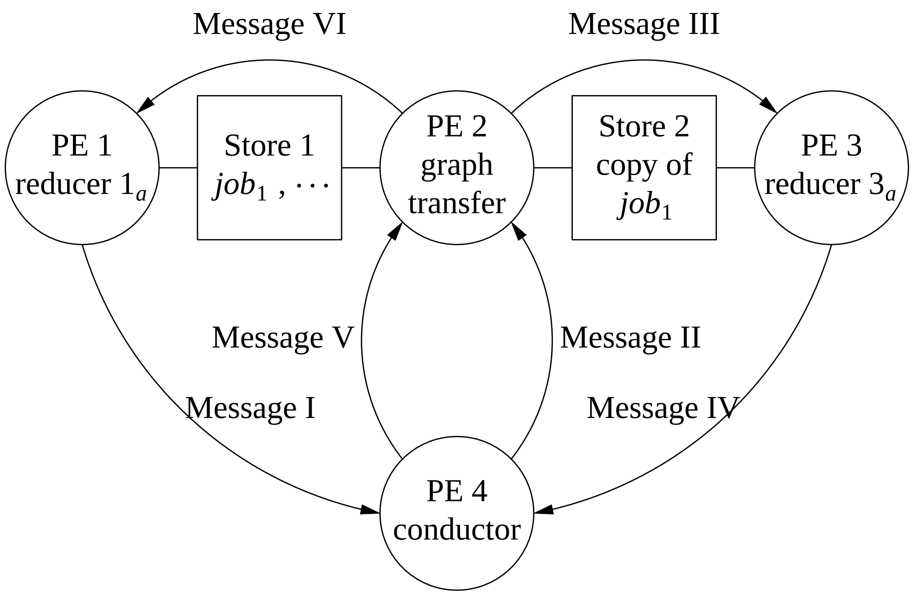


Figure 4 : Graph and message transport

Message I: Create jobs

Reducer 1_a on processing element 1 notifies the conductor of the creation of potential jobs located in store 1. The size of the graphs representing the jobs and the pointers to their roots are part of the message.

Message II: Transport job

The conductor decides to allocate reducer 3_a to the first job, and sends a message to the graph transfer process on processing element 2. The message contains the identification of the producer and the consumer of the graph representing the job and its location. Since processing element 2 has local access to both the source and the destination area, the graph can be transported node by node without requiring any intermediate copies. This advantage is due to both the use of overlapping stores and the separation of synchronisation and communication. The conductor has a good opportunity to exploit this property of the architecture in its allocation policy.

Message III: Start evaluation

When the transport has finished, reducer 3_a must be made ready. This can be accomplished by allowing the graph transfer process to pass information to the local scheduler of processing element 3. This form of synchronisation can not cause delays, since the receiving party is guaranteed to be waiting for it. The pointer to the root of the graph is part of the message.

Message IV: Result available

The availability of the result has to be announced to the conductor, since it must know when a reducer is free to receive a new job. The conductor also organises the transport of the result. The message contains the whereabouts of the result and the identity of its producer and consumer.

Message V: Transport result

The transport of the result is similar to job transport.

Message VI: Job complete

The scheduling administration on processing element 1 is updated, to register that a job that reducer 1_a is waiting for has now arrived. By the time that all outstanding jobs have been completed, the waiting reducer is made ready by the local scheduler.

A similar communication pattern emerges if jobs are to be transported under less favourable circumstances. The transfer will involve more processes and intermediate copies can no longer be avoided.

2.4. Mode of operation

We think of a parallel architecture for reduction as an embedded processor in a conventional host computer system. The operating system of the latter provides facilities to load and execute an application on the embedded system. The embedded processor is allocated to a single task, in the form of the main expression to be reduced, and remains allocated to the task until it completes. This obviates the need for multi-programming and other complications necessary in a general purpose system. We can even afford to omit support for input/output operations, because the embedded system may be fed a stream of jobs, which it will turn into a stream of results. While preparing the next job, the host may perform the necessary input/output operations.

Before an application can be started, its representation has to be prepared for execution. Depending on the way the reducer references the representation it may be (partially) preloaded in the processing elements, or could be transmitted as part of the jobs. If the demands with respect to the necessary code of the parallel computations are highly dynamic, preloading appears to be wasteful of both space and time. If the same code is required by all jobs, preloading is more economic.

The self modifying (sometimes called self optimising) property of the code generally used in graph reduction has a menacing characteristic to the code management scheme. Although semantically equivalent, some representations of the same function consume more space than others. Consider as an example, the function that computes the list of natural numbers. As soon as a certain number of elements of the list have been evaluated, the representation will have grown with respect to its initial form. Keeping the representation as it is saves time, when elements of the list are needed more than once. Reverting to the original form saves space, but requires the list to be recomputed if it is needed again. In a sequential graph reduction system, it may be expected, that the self modifying property may be controlled more easily than in an implementation where code is distributed over a network of processing elements. The reason is, that transportation of a large representation of a function incurs a time penalty with respect to a small representation. In the extreme case, it may even be worthwhile to perform an amount of recalculation to reduce communication costs and still achieve best performance. In our experiments we have selected the behaviour that gave the best performance improvement with respect to normal sequential versions of the same applications.

3. Performance model

To quantify the performance difference between sequential lazy graph reduction and graph reduction with the proposed parallel strategy and architecture, some measures are defined and applied to the application programs. With normal lazy graph reduction, the total execution time for a program is assumed to be largely dependent on the total number of reduction steps. If the individual reduction steps require roughly the same amount of computation, this relation is assumed to be linear. Such is the case with the combinator reduction system used in our experiments.[9] Therefore, the amount of work involved in normalising an expression is identified with the number of reduction steps involved. The definition of the sandwich strategy is such, that there is no difference in the total number of reduction steps required, whether a program is evaluated under the normal lazy strategy or with the sandwich strategy. Using the sandwich strategy, the net execution time of the program is less, due to parallel evaluation of jobs. The diagram of figure (5) schematises this difference. The horizontal line segments represent the number of reduction steps required by the different branches in the evaluation.

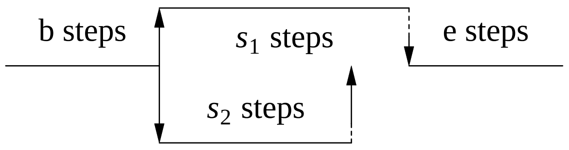


Figure 5 : Time diagram of parallel evaluation

On a system with unlimited processors and free data communication the total number of reduction steps, when *m* branches are generated, is:

$$R_t = b + \sum_{i=1}^{m} s_i + e \tag{1}$$

With the sandwich strategy the net number of reduction steps is:

$$R_n = b + \max_{i=1}^{m} s_i + e \tag{2}$$

The numbers b, $s_1 \cdots s_m$ and e in (2) are also interpreted as net reduction steps, rather than total reduction steps as in (1). The performance gain of parallel graph reduction over lazy graph reduction may now be expressed as R_t / R_n.

This is not a realistic approximation, since programs must be partially rewritten before the sandwich strategy may be applied effectively. Therefore, it is only fair to refer to the measure R_s, which gives the number of reduction steps for the sequential, untransformed version of the same program. The ratio R_s / R_n is considered to be a more realistic measure of performance gain. The ratio R_s / R_t gives the performance loss due to the cost of program transformations required to exploit parallelism.

Refinements are introduced to model some of the delays that may be experienced in the system. The first refinement compensates for loss in computing resources due to the transportation of jobs and results, since in the proposed architecture, the processing elements operate on private stores. In the modified time diagram of figure (6), the horizontal axis represents reduction steps as before. The length of a diagonal arrow represents the size of a graph that is transported, measured as a number of nodes. A graph transfer process behaves like a pipeline: one processor collects the nodes of the graph and sends a stream of nodes through the network. At the end of the pipeline a companion processor assembles the copy of the graph. In the general case two processors are actively working on the same transport. Transportation cost is expressed in reduction steps, by equating the time necessary for the transportation of T nodes with that spent in one reduction step. Furthermore, a penalty of C reduction steps accounts for the time spent in communication between processes. The roman numerals used to identify the transactions in figure (4) are shown in parentheses in figure (6).

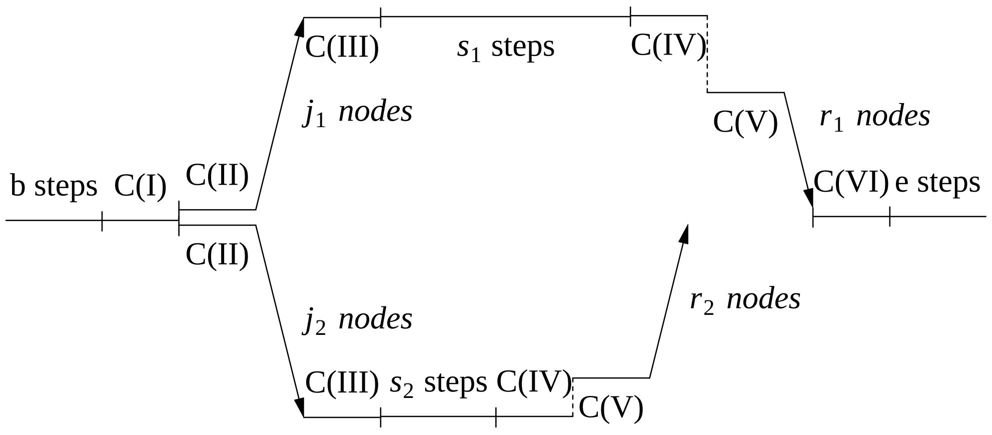


Figure 6 : Time diagram of parallel evaluation and transportation

The transportation cost is dependent on the distance travelled. As a first approximation, we would like to ignore locality, and assume that all graph transports relate to the same distance. The values of C and T are regarded as constants of the hardware and software configuration of a particular implementation of the proposed architecture.

In the performance measures developed thus far, the role of R_n is assumed by a new quantity R_g, which takes data communication cost into account. Let j_i and r_i represent the numbers of transported nodes in respectively the $i-$th job and the $i-$th result. The communication cost pertaining to the $i-$th job/result is:

$$c_i = \left\lceil \frac{j_i}{T} \right\rceil + \left\lceil \frac{r_i}{T} \right\rceil + 4\,C \tag{3}$$

The gross number of reduction steps of the whole family of m jobs is defined as:

$$R_g = b + \max_{i=1}^{m} (c_i + s_i) + e + 2\,C \tag{4}$$

The ratio $S = R_s / R_g$ gives the maximum speedup that can be attained. If the number of processing elements N required to achieve this speedup is taken into account, we find for the processor efficiency:

$$E = \frac{R_t}{R_g \times N} \tag{5}$$

The enumerator in (5) represents the amount of work done, whereas the denominator represents the maximum available computing capacity.

4. Optimal scheduling

Before considering the implementation of “on-the-fly” load balancing on our experimental reduction machine, we have investigated the consequences of the performance model outlined in the previous sections. This model assumes that the number of processors is sufficiently large to allow every job to be scheduled for execution as soon as it is generated. In the more realistic case of a limited number of processors, jobs will have to wait until a reducer becomes available. To calculate the best possible performance of an application on a given architecture, we have used the data obtained with the performance model to compute an optimal mapping of the generated jobs onto the available processors. This mapping, which minimises the turn around time, is called an optimum schedule. Computing an optimal schedule a posteriori serves two purposes. At first it yields an upper bound for the speed up that can be attained with the given application on the class of architectures considered. Secondly, an optimum schedule can be useful when the same application is executed frequently with different input data and when the generation of jobs hardly depends on the input data. This is the case with the fast Fourier transform, Wang's algorithm and the tidal model, provided the size of the problem remains fixed. For example, the latter application is designed to be used frequently and the generation of parallel jobs in the program only depends on geographical data, which are not likely to change often.

4.1. Scheduling of jobs

The illustration of figure (7) shows two jobs ($fork_1$ and $fork_2$) that have executed a *sandwich* primitive and three jobs that remain sequential (mid_1, mid_2 and mid_3). The horizontal axis represents the elapsed time as measured in reduction steps. The depicted durations of all job entities include the communication cost that is modeled by the parameters C and T in the performance model (shown by the dashed arrows).

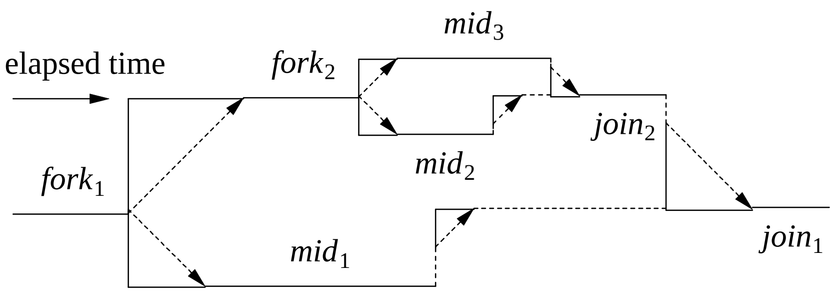


Figure 7 : Fork, mid and join jobs

After evaluating a sandwich reduction step, a job is suspended until the forked jobs have all terminated. From a scheduling point of view, this gives rise to three different job entities with a strict precedence relation:

fork jobs

A fork job executes a certain amount of reduction steps and then spawns a number of descendant jobs.

join jobs

When its descendants have terminated and their results have arrived, a fork job may resume reduction until it either terminates or encounters another sandwich application. In the first case the job is called a join job, the second case classifies it as a fork job again.

mid jobs

A mid-job does not execute the *sandwich* function and remains a sequential job.

Once an application has been run, all relevant data that is needed to compute the duration of fork, mid and join jobs is collected. The problem that remains to be solved in order to obtain an optimal schedule is to find a distribution of fork, mid and join jobs that satisfies the given precedence relations and minimises the total execution time.

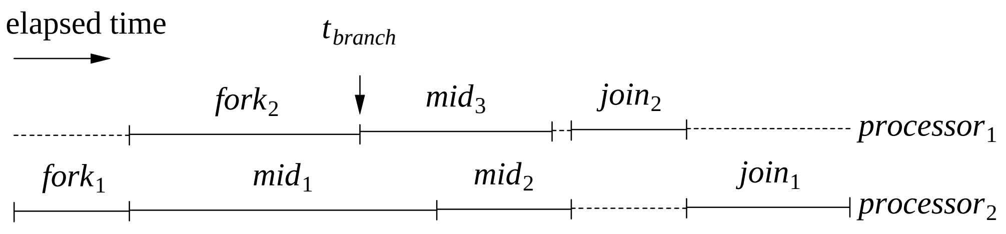


Figure 8 : An optimal schedule with two processors

As an example, figure (8) illustrates a schedule of the jobs involved in the application of figure (7) on a two processor system. The dashed lines represent the time periods that a processor is idle. When the job $fork_2$ wishes to submit its two descendant jobs (at $t = t_{branch}$), there exists a choice whether $processor_1$ should continue to execute job mid_2 or job mid_3. Both allocations represent a partial schedule and should be evaluated to decide which of the two is the shortest. The diagram of figure (8) shows the optimum schedule for this problem. In large applications many branches arise, yielding a vast search space to find the optimum schedule. The search for an optimal schedule with three types of jobs and prescribed precedence relations is an NP-complete problem.

4.2. Branch and bound algorithm

The algorithm that we have used to find the optimum schedule constructs a tree of possible allocations of jobs to processors. It is based on the branch and bound principle.[10] Each node in the tree represents the choice of allocating a job to a processor. A path from the root of the tree to a leaf forms a complete schedule. While the tree is constructed in a depth-first manner, an administration of available jobs is built and attached to each node of the tree. This is necessary because the set of available jobs at each node depends on the history (i.c. which fork jobs were executed). The fact that join jobs have to be scheduled at the same processor, where the corresponding fork job once was allocated also renders the allocation policy history sensitive. If this restriction on the allocation of join jobs would not have been imposed, the system would have to physically transport the representation of the join jobs to the elected processor. It is expected, that the incurred data communication cost does not outweigh the gain in scheduling efficiency that can be obtained by unconstrained allocation of join jobs. In our application programs and on our architecture, the cost to transport the representation of a join job is more than an order of magnitude larger than its reduction cost.

To reduce the size of the search tree, the scheduling program computes a lower bound on the best possible schedule that can be realised from a given node and compares this bound with the best schedule found so far. If the lower bound exceeds this schedule, the search beyond this point is cancelled. The lower bound is calculated with the expression $t + e/p$, where t is the elapsed time, measured in reduction steps, to arrive at the given branch point (e.g. $t = t_{branch}$ in figure 8). The quantity e represents the total number of reductions steps that remain to be performed in all jobs, from the current branch point until the end of the application. The ratio e/p equals the processing time required to execute the remaining amount of work (e) if an exact partitioning of the work over the available (p) processors would be possible. The lower bound coincides with the real optimum schedule, if this exact p-partition exists for the jobs that remain to be executed.

The proposed branch and bound algorithm is most effective if the search is directed in such a way that a near optimum solution is found quickly. If such a near optimum is established in the very beginning of the schedule, many search paths in the remainder of the program representing longer schedules can be effectively pruned. To achieve this, the following heuristics have been incorporated in the program:

a) Because in our applications join jobs always contain a negligible amount of work, first an optimal schedule is computed for fork- and mid jobs.

b) If a choice exists, a fork job has priority over a join job, thus fork jobs are scheduled first. Scheduling a fork job increases the number of jobs that still have to be scheduled, while allocating a mid job decreases this number. The heuristic assumes, that better schedules arise if more jobs are available.

c) A larger job takes priority over a smaller job. This heuristic has been proven to yield a schedule that is at most a factor of two larger than the optimal schedule.[11]

4.3. A parallel program to find the optimum schedule of a set of jobs

While designing the program to find optimal schedules for divide-and-conquer algorithms, it appeared that the program itself could be written as a divide-and-conquer application and included in the set of application programs that we use to test our parallel reduction model. However, because jobs have to be self contained, a central administration containing the best schedule found so far, can only be maintained at high cost. This implies that the pruning of subtrees can not be performed. The gain in scheduling time due to parallel evaluation has to be compared to the loss in search efficiency. Considering that the search with heuristics and lower bound comparison only realises a speed up by a factor of two with many jobs of about the same size, the speed up of the scheduling algorithm by parallel execution soon exceeds the loss in search efficiency. The threshold mechanism that we use for dynamic grain size control causes all mid jobs to be of approximately the same size.

The SASL function *Alloc* of figure (9) implements the tree search algorithm without the lower bound calculation and cancelling of subtrees. This parallel version of *Alloc* shows how the sandwich function is used in combination with the threshold mechanism. The *Alloc* function is a simplified version of the result obtained by the job-lifting and grain size transformations, which are described in part I of this paper.

```
 1.  Alloc  jobold  jobnew  procold ( ) level
 2.       = Process (jobold ++ jobnew) procold  level
 3.  Alloc  jobold ( ) procold (proc : procnew) level
 4.       = Alloc ( ) jobold (proc : procold) procnew  level
 5.  Alloc  jobold  jobnew  procold (proc : procnew) level
 6.       = Busy  proc          → allocnextproc
 7.         jobnew = ( )        → allocnextproc
 8.         level > Threshold → (allocjob$_1$  nextlevel) : (allocjob$_2$  nextlevel)
 9.         sandwich  cons (allocjob$_1$ nextlevel ) (allocjob$_2$ nextlevel )
10.         WHERE
11.         jobold$_1$ : (job : jobnew$_1$) = FindNextJob  jobold  jobnew  proc
12.         allocnextproc = Alloc ( ) (jobold ++ jobnew) (proc : procold) procnew  level
13.         allocjob$_1$  = Alloc ( ) (jobold$_1$ ++ jobnew$_1$)
14.                                    ((Allocate job proc) : procold) procnew
15.         allocjob$_2$  = Alloc (job : jobold$_1$) jobnew$_1$  procold (proc : procnew)
16.         nextlevel = level + 1
```

Figure 9 : The parallel tree search function

The function *Alloc* scans two administrations: a job administration *jobold* ++ *jobnew* and a processor administration *procold* ++ *procnew*. The lists *jobold* and *procold* contain jobs and processors that have already been scanned, whereas *jobnew* and *procnew* contain the items that have not yet been considered. The heads of *jobnew* and *procnew* are the job respectively processor that are currently considered for allocation. The applications of $allocjob_1$ and $allocjob_2$ (in lines 8 and 9) constitute the two alternatives of allocating the actual job to the actual processor ($allocjob_1$) and not allocating the actual job ($allocjob_2$). The latter alternative causes the next job to be considered for allocation. Both alternatives are submitted for parallel evaluation by the sandwich application in line 9. However, this line is only executed if the actual depth of the tree (*level*) is below a certain value *Threshold*. If the *level* exceeds the threshold value, the same alternatives are evaluated in line 8, but in this case sequentially.

The definition in line 1 applies if *procnew* is empty, which means that no more processors are available for allocation. The function *Process* advances the time until one of the processors becomes free (via termination of the current job allocated to that processor). *Process* then recursively calls *Alloc* to perform allocation of the recently freed processor(s). The definition of line 3 applies if *jobnew* is empty, which is the case when no more jobs are available for allocation to the current processor. However, there may still be join jobs that are ready for execution and have been skipped because they have to be executed by a different processor. Thus instead of terminating, the function *Alloc* is called recursively in line 4 to enable join jobs to be allocated to the next available processor. The function *Busy* in line 6 checks if the current processor is ready to receive a job. The function *FindNextJob* in line 11 scans the job administration *jobnew* for the next job that is both ready and allowed to execute on processor *proc* (join jobs are preallocated). Jobs are found in a sequence that satisfies the heuristics b) and c) of the previous section. Skipped jobs are prepended in front of *jobold*, such that the result $jobold_1 : (job : jobnew_1)$ is still the complete administration and *job* is the required next job.

5. Results

Having developed annotated parallel applications, a basic concept of a parallel architecture, a performance model and an algorithm to calculate optimal schedules, we can now present preliminary results. The most important result is the speedup that may be attained with the various applications. The data that the scheduling algorithm requires to compute the speedup could be obtained by running the applications through a fully implemented parallel reduction machine. However, since the job structure that develops during execution of the applications is strictly hierarchical, we were able to extract the required data from a simple pilot implementation. The remainder of this section describes the experimental system that we built and the way the performance figures were obtained from the experiments.

5.1. Experiment

The experimental system consists of a alternating string of processing elements and overlapping stores.[5] By limiting the maximum depth of the job-tree to the number of processing elements, we were able to test our ideas while the design of the conductor is still in progress. Currently, a processing element supports one reducer and one graph transport process. During an experiment, the first processing element in the string receives the main expression. The jobs produced from the main expression are evaluated one by one on the second processing element, which in turn may pass jobs it creates on to the third processing element etc. This corresponds to a pre-order traversal of the job-tree. It does not however cause reduction to be performed in parallel. A run on the experimental system produces the data that the optimal scheduling algorithm requires to compute the speedup that may be attained. In a full implementation of our reduction machine similar data would be exchanged between reducers and the conductor to perform on-the-fly scheduling.

5.2. System parameters

The measurements on the experimental system have been performed using a slow, fixed combinator graph reducer.[9] The observed data communication performance of 10000 nodes per second is based on the binary node representation of this reducer (one node occupies 6 bytes of storage). To obtain a realistic estimate of the T-factor we should use the real-time performance of an optimised sequential combinator graph reducer,[12] which exceeds 10000 reduction steps per second on a VAX 11/750. Experience with CPU bound applications has shown that the MC68010 processors of the experimental system have about the same performance. The reported reduction speed can be improved by one order of magnitude via optimisation techniques, but the same holds for the data communication speed via the use of special hardware. The latter may even yield an improvement of two orders of magnitude (see section 2.2.3.3).

Considering both performance figures we may derive a value for T = *nodes per second* / *steps per second* = 10000 / 10000 = 1 nodes/step. Tuijnman and Hertzberger[13] report message passing delays on a multi processor system that is similar to ours. When two processors are connected by a shared memory, which is the case for communication between the conductor and reducers, a delay of 2 msec is found. Therefore a reasonable value for C = *steps per second* $\times$ *seconds* = $10000 \times 0.002 = 20$ steps.

5.3. Applications

A set of five application programs has been run on the experimental system to acquire the data needed to perform optimal schedule calculations. Four of these application programs; quick sort, the fast Fourier transform, Wang's partition algorithm and the tidal model have been discussed in part I of this paper. Particular attention has been paid to annotation and transformation to adopt the applications to parallel execution. In this part of our paper we introduced a fifth application, that calculates the optimal schedule of a set of jobs with hierarchical precedence relations. The remainder of this section presents a brief description of the input data it has been provided with, followed by a discussion on performance characteristics under optimal scheduling conditions.

5.3.1. The optimal scheduling application

The scheduling program presented in section 4.3 has been applied to (artificial) performance data of seven hypothetical jobs. As such the program can be executed like any other parallel application and the acquired data can be used to calculate optimal schedules and maximum speed-up figures. To study the performance of an annotated program on a parallel architecture, the relation between four architectural variables needs to be considered.

Speed-up factor

This quantity is defined as the quotient of R_s (the execution time of the sequential program) and the duration of the optimal schedule. It corresponds to the intuitive notion that is conveyed by the word speed-up and it is the objective function that has to be maximised. The limit of the speed-up when the number of processors goes to infinity is the quantity S.

Threshold value

A threshold is present in three of the five application programs (see figure 9). In these programs an abundant amount of parallel jobs is generated by recursive function calls. A comparison with the threshold parameter stops the recursion when the grain size of the jobs becomes too small. For a given application size and a given number of processors an optimal value for the threshold is determined. The threshold value is optimal when the speed-up is maximal.

Synchronisation and communication costs

These are the parameters C and T of section 5.2. Their value determines the minimum grain size of a job that can still be submitted for parallel execution without decreasing the overall speed-up.

The number of processors

This parameter can be varied to determine for a given application the smallest value for which the maximum speed-up can be achieved. Another possibility is to determine the maximum number of processors for which the efficiency E of the system stays above a certain cost-effective value.

To present the performance data of the scheduling application, two sets of curves are drawn in figures (10) and (11). In both figures the speed-up is plotted against different values of the threshold. For the scheduling application the threshold value represents a specific depth in the search tree beyond which no more parallel jobs are generated. At the left end of the x-axis in the figures this depth is zero, which means that no parallel jobs are submitted. Increasing the threshold value by one means doubling the number of parallel jobs, as long as the search tree remains balanced.

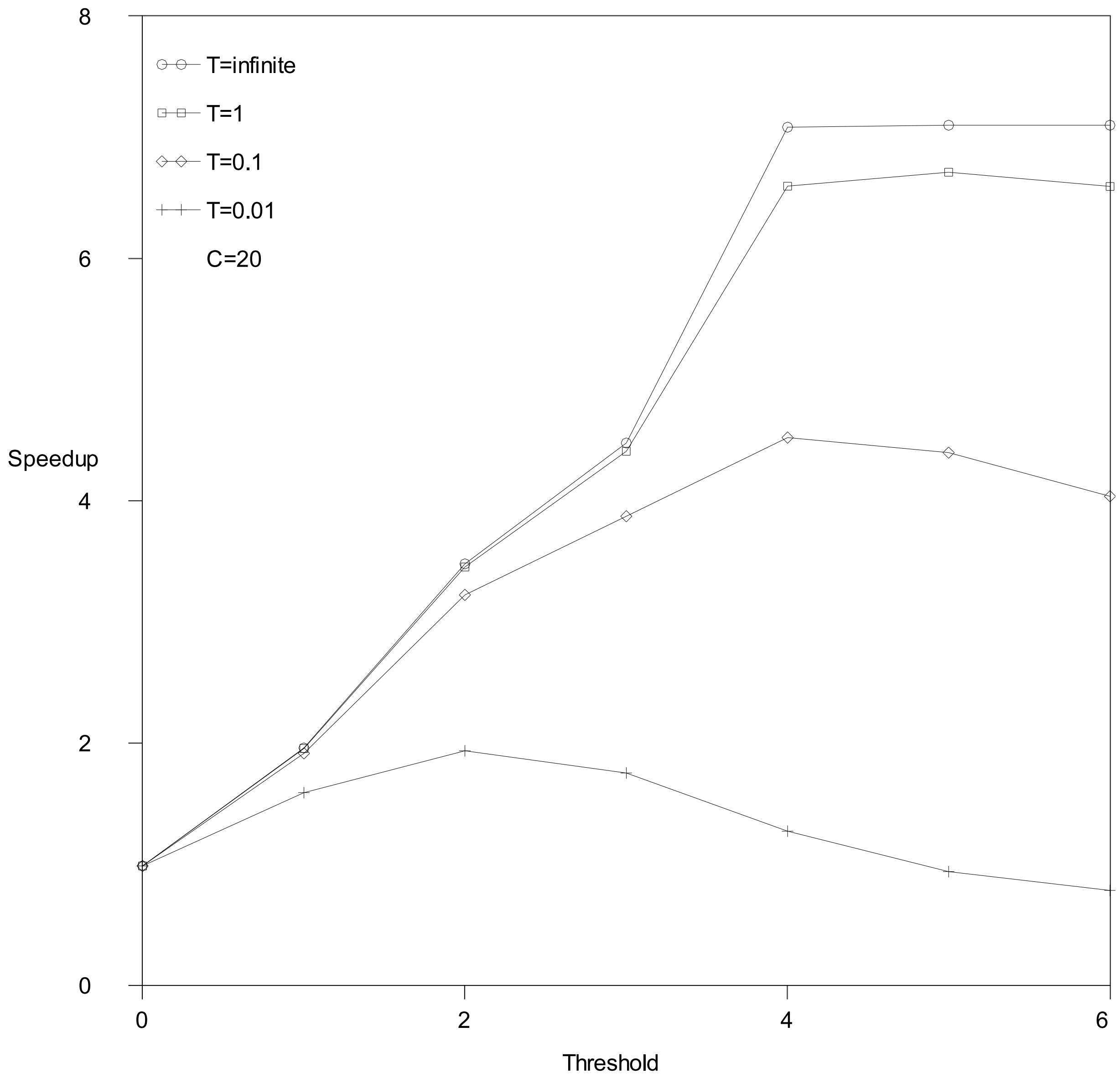


Figure 10 : Speedup of the scheduling program for 8 processors with various T-factors

In figure (10) different speed-up curves are shown with the number of processors fixed to eight. Each curve corresponds to a certain performance of the data communication subsystem, expressed by the T-factor belonging to the curve. The figure shows that for this application the T-factor should not drop below a value of 0.1 (i.e. the required throughput of the communication network should be higher than 1 node per 10 reduction steps). With this throughput a maximum speed-up of 4.6 can still be achieved with an optimum threshold value of 4. It is assumed, that the lowest acceptable processor utilisation is 50% (a speed-up of 4.6 with 8 processors in this case). Figure (10) also shows that data communication becomes a negligible factor when the network throughput exceeds the value of one node per reduction step ($T = 1$). The performance data of the other application programs show a similar behaviour. In all cases the network throughput has a critical region between $T = 1$ and $T = 0.1$.

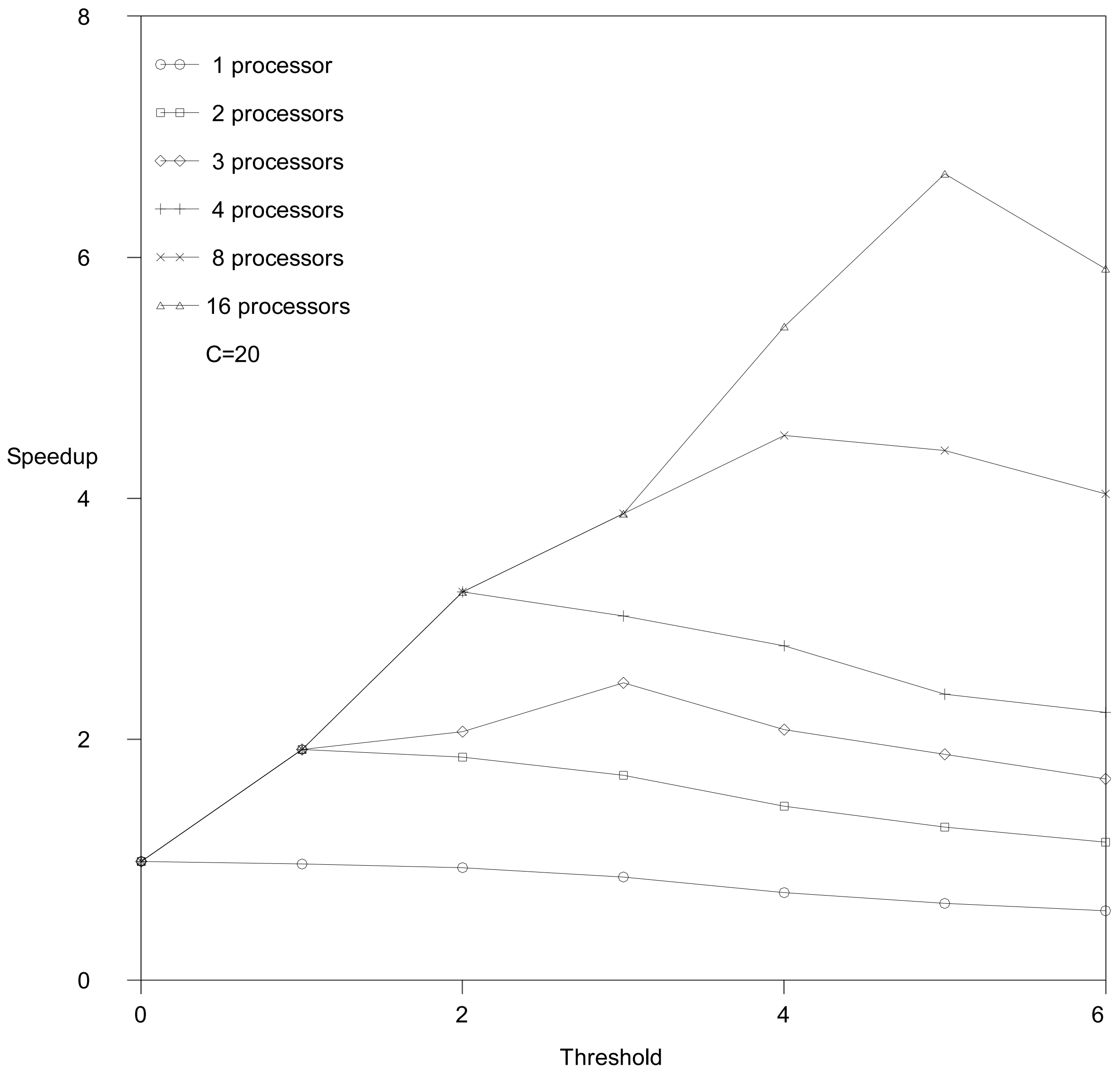


Figure 11 : Speedup of scheduling for $T = 0.1$ with various numbers of processors

Figure (11) shows a set of speed-up curves for the lowest acceptable network throughput of $T = 0.1$. For each number of processors an optimum value of the threshold exists and the corresponding processor utilisation decreases when the number of processors increases, to drop below the assumed acceptable limit of 50% for 16 processors or more. We may conclude that the scheduling application with the given input and the given data communication system with ($T = 0.1$) can have an economical speed-up of 4.6 with 8 processors.

5.3.2. Optimal performance

To calculate the optimal schedules for the remainder of our application programs, they have been supplied with the following input data. The quick sort function has been applied to a list of 1024 values, resulting from the sine function applied to the first 1024 natural numbers. The fast Fourier transform algorithm calculates the frequency and phase spectrum of a real valued function in the time domain. The parallel version of the algorithm has been supplied with a data array of 512 elements, containing 8 periods of a sawtooth wave form with an amplitude of 64. The real part of the 512-point transform shows peaks of the same height at every eighth point, corresponding to the flat frequency spectrum of a sawtooth. The input for the Wang algorithm was a square, diagonally dominant, tri diagonal matrix of 255 rows. The tidal model has been run on a grid of 10×10 points representing an area of 1000 km^2, during 5 time steps of about 15 minutes simulated time. The initial conditions were set to an average water depth of 30 metres and a slope in the water height of 3 metres in the x direction.

The best economical speedup for the application programs is presented in table (1). The first row gives an impression of the order of complexity that ranges from O(n) to O(n!). The second row states the execution time (R_s) of the sequential versions of the applications on the given input data. The third row shows the performance gain or loss (R_s / R_t) that is incurred by transforming the programs into a form suitable for parallel evaluation. The inclusion of a threshold mechanism and the addition of the *sandwich* and *own* functions are responsible for most of the performance loss. In case of the tidal model the transformation is particularly complicated. It involves the introduction of streams to model concurrent processes and the division of a space staggered grid into equal parts. The resulting program appears to be a more efficient version of the original program. We have not been able to find an explanation for this phenomenon. Table (1) presents the results of the tidal model in case of a bisection of the grid.

The fourth row in table (1) presents the best speedup results that can be obtained with the given application and a minimum T-factor (fifth row), provided that the processor utilisation does not drop below the supposed economically acceptable value of 50%. The minimum T-factor represents the data communication capacity that should at least be available to achieve the given speedup. The next two rows show the optimal values of the threshold and the number of processors that should be used under these circumstances. The penultimate row of the table gives an estimate of the number of nodes that is needed by the most heavily used processor; all other processors need fewer nodes. These estimates are based on a reducer that uses fixed size nodes (each node has a tag and two pointer fields) and a reference counting garbage collector. With a non-reference counting garbage collector at least twice the estimated amount of store is necessary to prevent garbage collection from requiring to much processing time.[5]

The last row of table (1) presents the maximum speedup (S) that will result if the communication performance (T) grows to infinity and unlimited processors are available. The values shown are based on the lowest threshold that we have used in the experiments. Comparing this row to the speedup figures shows that much of the potential available parallelism can be exploited on a practical local memory architecture. The maximum speedup may be larger than the number of processors used because the speedup refers to the sequential untransformed versions of the programs and the transformation by itself may already speedup computations.

Legend	schedule	quick sort	fast Fourier	Wang	tidal model
Program transformation					
Order of complexity	$n!\,/\,p!$	$n \log n$	$n \log n$	n	n^2
Sequential steps	530908	493205	262655	190930	199644
Transformation loss	0.98	0.94	1	0.87	1.29
Best economical schedule					
Economical speedup	4.6	2.2	4.5	2.7	2.2
Minimum T-factor	0.1	1	1	1	0.1
Threshold	4	32	128	-	-
Number of processors	8	4	8	5	2
Minimum space per processor (nodes)	7995	8223	5506	6213	5614
Unlimited processors and no data communication cost					
Maximum speedup	15.2	2.8	7.4	3.7	2.5

Table 1 : Optimal performance of the five application programs

The Wang partitioning algorithm solves a set of linear equations that result in a tridiagonal coefficient matrix. Because this algorithm has been designed for parallel execution and as a consequence lacks a sequential counter part, the transformation loss has to be interpreted differently. The execution time (R_s) of the Wang program applied to an undivided matrix has been compared to the total number of reduction steps when the program is applied to the same matrix divided in five equal parts. The Wang algorithm and the tidal model have been annotated in such a way that a fixed number of jobs is generated during execution. This number is determined by the transformation. The reason for doing so is that the grain size of the jobs does not depend on the input data and can be fixed by the programmer. Both quick sort and the schedule program generate jobs whose grain size depends on the calculations. In such cases the number of jobs can not be fixed a priori and a threshold mechanism has to be included by the programmer. In case of the fast Fourier transform the number of jobs does not depend on the calculations and the grain size of jobs could be fixed by a transformation into a program without a threshold mechanism. However, due to the nature of the calculations a recursive version of the algorithm with a threshold mechanism is much simpler to derive.

6. Conclusions

Parallel graph reduction based on jobs is a useful concept. It allows divide-and-conquer applications and programs based on synchronous communicating processes to run faster on a parallel machine. The architecture of such a machine can be based on local store. Jobs are copied from one processing element to another, but work is not duplicated. Even cyclic programs can be made to benefit from parallelism on an architecture that does not support globally cyclic graphs.

Centralised control over the machine by the conductor is feasible because the interaction between the applications and the central control is restricted to a minimum. The threshold mechanism that we propose to regulate the grain size of parallel computations serves to restrict such communications. Centralised control is also effective. The information about the behaviour of the running application that is available to the conductor enables it to schedule jobs in a near optimal way. At each decision point the conductor has knowledge about the resource requirements of the set of jobs that are currently being offered for consideration as parallel grains. As soon as the system is sufficiently loaded with jobs, new requests may be refused, to prevent the administration from overflowing.

The job concept causes the process structure of a parallel computation to be strictly hierarchical. This makes high speed data communication possible, since the transport of a job or result can be separated from synchronisation. The transactions with the conductor involve small messages, which are transmitted when synchronisation occurs. The space to store these small messages is always available because the transmit operation is blocking. The jobs and results transmitted after consulting the conductor, may contain a much larger volume of data that can be transmitted without further synchronisation. In this case the space to store the message at the receiver side is reserved before the transaction is started. Job and result transport is simple enough to be implemented directly in hardware, allowing for a data communication speedup of two orders of magnitude with respect to a software implementation. The separation of synchronisation and communication in general purpose systems is not feasible since one can not always afford to have both the producer and the consumer of a message to be delayed while data is being exchanged. Another difference between ours and a general approach to concurrency is that reducers may be considered both as client and as server. Hence a request for service may safely be refused because the client is capable of servicing its own request. The cost of such a refused request is merely the time necessary to send a message to the conductor and wait for the reply. The actual job graph is not transmitted in that case.

The results that we have presented are based on a posteriori optimal scheduling. Rather than building a full scale system, we have restricted ourselves to a pilot implementation. The scheduling data are recorded during the run of an application and processed after the application has been run. The assumptions about the number of available processors are realistic, but the parameters and relations that model the data communication network are a first approximation that will be refined in future work. Two other differences with scheduling as it would be performed on a fully implemented system are the accuracy of the parameters that determine the decision making policy and the time that the scheduling algorithm is allowed to spend on making a decision.

We have shown that under conservative assumptions with respect to the performance of the data communication sub-system there is a situation where a processor utilisation of over 50% may be attained. Some applications are more critical in this respect than others because their computational complexity is lower in terms of the job and/or result size. The actual number of processors that may be occupied depends on the application and the problem size. An interesting aspect of our proposal is that we have managed to escape from the computer science tradition that a new compiler should compile itself. Instead the heart of our system is used as one of its applications.

Acknowledgements

We gratefully acknowledge the support of the Dutch parallel reduction machine project team. Rutger Hofman made valuable comments on draft versions of the paper.

References

1. W. G. Vree and P. H. Hartel, "Parallel graph reduction for divide-and-conquer applications -- Part I: program transformation," PRM project internal report, Dept. of Comp. Sys, Univ. of Amsterdam (Dec. 1988).
2. H. H. Wang, "A parallel method for tri-diagonal equations," *ACM transactions on mathematical software* **7**(2) pp. 170-183 (Jun. 1981).
3. R. F. H. Hofman, "An on-the-fly scheduling algorithm for an experimental parallel reduction machine," PRM project internal report, Dept. of Comp. Sys, Univ. of Amsterdam (May 1988).
4. J. Cohen, "Garbage collection of linked structures," *ACM computing surveys* **13**(3) pp. 341-367 (Sep. 1981).
5. P. H. Hartel, *Performance analysis of storage management in combinator graph reduction,* Dept. of Comp. Sys, Univ. of Amsterdam (Oct. 1988). Ph.D. thesis
6. D. A. Turner, "Functional programs as executable specifications," pp. 29-54 in *Mathematical logic and programming languages,* ed. C. A. R. Hoare and J. C. Shepherdson, Prentice Hall, London, England (Feb. 1984).
7. W. G. Vree, "Parallel graph reduction for communicating sequential processes," PRM project internal report, Dept. of Comp. Sys, Univ. of Amsterdam (Aug. 1988).
8. R. B. Kieburtz, "The G-machine: {A} fast, graph-reduction evaluator," pp. 400-413 in *2nd Functional programming languages and computer architecture, LNCS 201*, ed. J.-P. Jouannaud, Springer-Verlag, Berlin, Nancy, France (Sep. 1985).
9. D. A. Turner, "A new implementation technique for applicative languages," *Software−practice and experience* **9**(1) pp. 31-49 (Jan. 1979).
10. N. J. Nilsson, *Problem solving methods in artificial intelligence,* McGraw-Hill, New York (1971).
11. R. L. Graham, "Bounds for certain multiprocessing anomalies," *Bell system technical journal* **45**(9) pp. 1563-1581 (Nov. 1966).
12. P. H. Hartel, "Performance of lazy combinator graph reduction," PRM project internal report, Dept. of Comp. Sys, Univ. of Amsterdam (Aug. 1989).

13. F. Tuijnman and L. O. Hertzberger, “A distributed real-time operating system,” *Software−practice and experience* **16**(5) pp. 425-441 (May 1986).