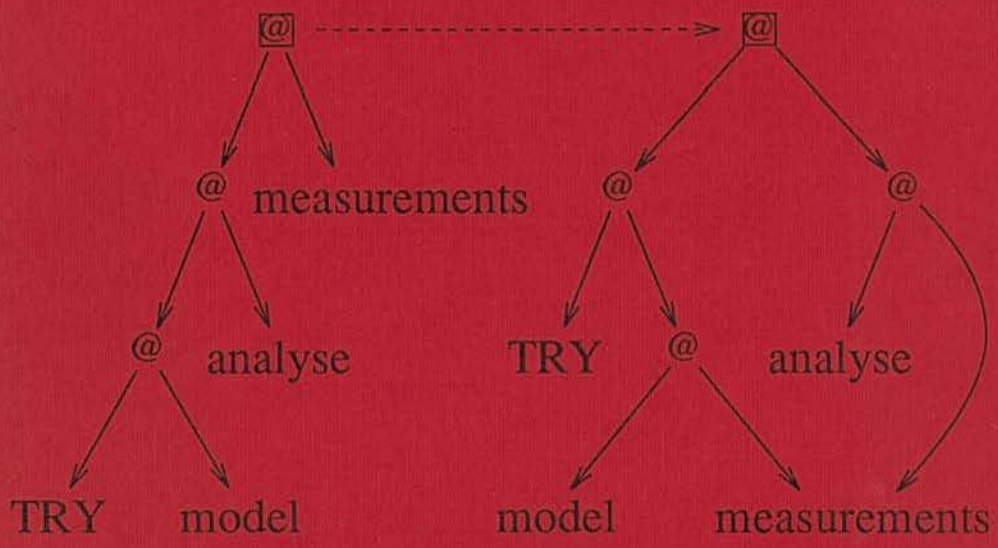
Performance Analysis of Storage Management in Combinator Graph Reduction

Pieter H. Hartel



Stellingen behorende bij het proefschrift van Pieter H. Hartel
Performance analysis of storage management in combinator graph reduction

(1) De toepassing van Schönfinkel's combinatoren bij de implementatie van een functionele programmeertaal is wegens zijn eenvoud aan te bevelen.
[D. A. Turner, *A new implementation technique for applicative languages*, Software practice and experience Vol. 9, No. 1, pp. 31-49, January 1979]

(2) Bij het gebruik van functionele programmeertalen is de semantische gap tussen algoritme en programma kleiner dan bij imperatieve talen.
[R. J. M. Hughes, *The design and implementation of programming languages*, PhD. thesis, Oxford University, 1983]

(3) Er is veel meer regelmaat aanwezig in de grafen die ontstaan tijdens combinator-graafreductie volgens Turner's methode dan men in eerste instantie zou aannemen.
[Dit proefschrift, eerste artikel]

(4) Een naïef hanteren van de "space/time trade off" kan aanleiding geven tot foutieve afwegingen, omdat het toevoegen van meer geheugen een systeem soms langzamer maakt.
[Dit proefschrift tweede artikel]

(5) Men kan niet stellen dat parallelle machines alleen geschikt zijn voor problemen die van nature parallellisme vertonen, zonder op de natuur van die problemen in te gaan.
[L. O. Hertzberger in *Architectuur: Von Neumann is dood, leve Von Neumann*, Intermediair, 11 maart 1988.]

(6) Turner's combinators geven niet noodzakelijkerwijs aanleiding tot fijnkorrelig parallellisme.
[Dit proefschrift, laatste twee artikelen]

(7) Bij talen zoals Algol-60 en Modula-2 zijn in- en uitvoerfaciliteiten niet opgenomen in de eigenlijke taal definitie maar in de runtime support. Dit maakt programma's geschreven in die talen juist minder overdraagbaar.
[P. H. Hartel, *Le langage Modula-2, concepts et expérience* Forum sur la micro informatique en physique nucléaire et physique des particules, ed: G. Fontaine, Collège de France Paris, Sep. 1983, pp. 323-331]

(8) Het essentiële verschil tussen Input-Output Tools en een gangbare parser generator zoals yacc is, dat voor tools een breadth first parser nodig is.
[J. van den Bos, M. J. Plasmeijer, P. H. Hartel, *Input-output tools, A language facility for interactive and real-time systems*, IEEE Transactions on software engineering, Vol. SE-9, No. 3, May 1983, pp. 247-259]

(9) Onderzoekers die artikelen opsturen naar refereed conferenties dienen bedacht te zijn op een beduidend mindere kwaliteit van de daaruit voortkomende referee-reports in vergelijking met de refereeing van vaktijdschriften.
[Het verslag van een willekeurige ACM conferentie]

(10) De controverse over het gebruik van het woord *programme* versus *program* in Engelstalige teksten is te beslechten op basis van de volgende stellingname: Uit citaten in de Oxford English Dictionary blijkt dat tot de 19e eeuw *program* de normale spelling was. Deze spelling bleef gehanteerd door o.a. Scott, Carlyle, Hamilton, en is te prefereren omdat ze conform is aan de gebruikelijke spelling van het Griekse *gramma* in woorden als *anagram*, *cryptogram*, *diagram* en *telegram*
[Fowler modern English usage, Oxford University Press, 1952]

(11) Het in alle details plannen doet sterk afbreuk aan de essentie van wetenschappelijk onderzoek.
[A. Lagendijk, *Niemand kan een ontdekking voorbereiden*, NRC Handelsblad, 2 november 1987]

(12)

Schiermonnikoog paradijs voor PRM project

Schiermonnikoog heeft zich de laatste dagen ontpopt als een ideaal oord voor langlaufers. In de duinen was zelfs een afdaling voor alpine-skiërs mogelijk. Een ondernemende fietsenverhuurder is op het autovrije eiland in een gat op de wintersportmarkt gesprongen; hij verhuurt ski's. In het noorden en oosten van het land zijn er volop langlaufmogelijkheden. (Foto ANP)

Performance Analysis of Storage Management in Combinator Graph Reduction

Performance Analysis of Storage Management in Combinator Graph Reduction

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 1 februari 1989 te 15.00 uur

door

Pieter Hendrik Hartel

geboren te Heemstede in 1953

1988

Prestatie-analyse van geheugenbeheer in combinatorgraafreductie

Promotores: prof. dr. L. O. Hertzberger
prof. dr. H. P. Barendregt

Faculteit: Wiskunde en Informatica

Printed at the Centrum voor Wiskunde en Informatica, Amsterdam

This work is supported by the Dutch ministry of Science and Education, dienst Wetenschapsbeleid.

à mon fils Olivier Nicolas

Table of contents

FIVE PAPERS

Preface

This thesis has two subjects: performance analysis of functional programs and the design and evaluation of a parallel reduction machine. Our primary concern has been to measure, compare, model and control the way at which the rapidly changing graph that represents a functional program during execution can be stored. We have measured the size, growth, shape, composition and structure of the graphs that arise during the execution of a benchmark of functional programs. We compare the performance of several garbage collection methods and analytically model the amount of garbage that is produced during reduction. In our parallel architecture we are able to control the way garbage is produced such that garbage collection can be performed locally by each processor in the system.

The work reported here has been carried out as part of the Dutch parallel reduction machine project. The purpose of the project was to answer the question: is it feasible to build a parallel reduction machine? Before we explain what the contribution of the thesis is to the answer we briefly outline the organisation of the project, because that gives the background for some of the decisions we had to take. (An introduction to the terminology of functional programming and reduction is provided in chapter 2).

In the project a dozen people from research groups of three Dutch universities (Utrecht, Nijmegen and Amsterdam) worked in close collaboration. The team made three important decisions. The first is to separate reduction and parallelism. The processes that perform reduction are denied knowledge of each others existence and parallelism must be managed separately. The development of efficient sequential reduction methods and methods to exploit parallelism are thus strictly separated and could be studied independently. An important consequence of this decision is that it practically rules out the exploitation of fine grain parallelism, because reducers are not allowed to communicate directly with each other. An advantage is that advances in sequential reduction technology are directly beneficial to a parallel machine based on the separation of concerns. The second decision is not to rely on the use of a common store for reduction. Instead each reducer is given a reasonable amount of private store, which does not make life harder on the reducers since they do not know of each others existence. Because there is no common store, management of parallelism must rely on a network that serves to connect the components of the architecture. The second decision also implies the exploitation of coarse grain parallelism. The third decision is to develop a reduction model that can be used as an intermediary between functional languages and reducers. This makes it possible to compare languages and their

implementations and to prove the correctness of implementations.

The work was divided into four complementary subprojects. The subdivision could be made because of the three decisions taken.

(1) Develop a reduction model that encapsulates both essential aspects of functional languages and essential concepts of their implementation. This model is to be used to compare theoretical and practical properties of reduction.

(2) Construct an efficient implementation of the reduction model on a sequential machine. This implementation must provide at least the same performance as for instance an implementation of the programming language Pascal.

(3) Investigate how functional programs can be annotated to obtain coarse grain parallelism during their execution. It must be possible to execute annotated programs efficiently on an architecture without a common store.

(4) Build an experimental parallel reduction machine that can serve as a test bed for the implementation of reduction models with annotations for parallelism.

The efforts of the project team and other researchers in the same area indicate that it is feasible to build a parallel reduction machine. In this thesis we present the design of a coarse grain parallel reduction machine that does not require a common store. Hand in hand with the design we develop a set of application programs that will benefit from parallel evaluation on the machine. We show that the class of applications with this benefit comprises divide-and-conquer algorithms and indicate how to structure the application programs. We describe the implementation of the machine that we used to evaluate the design, using the parallel application programs as a benchmark. The evaluation shows that it is possible to speedup execution of a substantial class of functional programs by exploiting parallelism.

Due to the division of responsibilities within the project, no attention is paid in this thesis to important issues such as the development of new compilation techniques for functional programming languages. In the area that was our responsibility (points 3 and 4 above) topics such as load balancing, networking and performance modelling and monitoring are not well developed yet. The refinements made by several new staff to the proposed design and the model used to evaluate it look promising. The momentum the Dutch reduction machine project has helped to accumulate guarantees that research in functional programming and reduction machines will continue.

Amsterdam
October 1988

Acknowledgements

Thanks are due to the members of the Dutch parallel reduction machine project team. They have created the environment in which the research described here could be carried out. Many of the ideas explored in this thesis were born during discussions and arguments with the other team members. Special mention should go to Wim Vree and Arthur Veen. They have spent a lot of time and effort in co-authoring three of the papers reproduced in the thesis. The help and guidance of Henk Barendregt, Bob Hertzberger and Rinus Plasmeijer was greatly appreciated.

We have used the functional programming language SASL by Prof. D. A. Turner to implement our application programs. Our experimental parallel imlementation of SASL was programmed in Modula-2, using the compiler that was written by Marius Schoorel.

I am grateful to D. Zwarst and his colleagues of the Centrum voor Wiskunde en Informatica for printing the thesis.

INTRODUCTION

Chapter 1

Storage management for functional programs

During the execution of a functional program much time is spent in storage management. This is a problem because the more time a system spends in storage management the less it does useful work. In this monograph we study the problem in two different contexts. The first is a sequential implementation of a functional programming language based on combinator graph reduction (see chapter 2 for an introduction to the terminology). The second context is a parallel implementation of the same programming language, where reduction and parallelism are carefully separated such that they can be studied independently. We have chosen a relatively simple implementation of reduction because it is sufficiently representative for state-of-the-art implementations of functional programming languages. In both contexts storage management involves allocation of new cells on request and the collection of garbage cells. Allocation is much easier to implement efficiently than garbage collection, because allocation is explicit. Garbage collection requires searching of the often cyclic graphs that are used to represent functional programs during execution. Many methods have been proposed to perform garbage collection, but the mutator (the process that produces garbage) is less often studied.[1] Our approach studies the mutator in relation to garbage collection. For instance we have found that for a benchmark of functional programs, the life time of most cells is short. This property can be used to make execution faster by organising the implementation of the programming language such that short lived cells are recuperated more rapidly than long lived cells. In other contexts similar results have been reported and used to an advantage.[2]

The phenomena that we have collected information about here concern the use of the store as a means to represent functional programs. The set of cells that represent a functional program under execution forms a rapidly changing graph. Storage cells contain data pertaining to the mutator as well as references to other cells. A cell may refer any other cell, including itself. We use a well defined mutator (a combinator graph reducer) and a set of application programs that we assume to be representative. We have studied various phenomena, such as the life time of cells, the rate at which cells become garbage and the average number of references to cells, that collectively give an impression of how the store is used. For two aspects of the interaction between the mutator and the collector a formal analytical model is developed. The experimental results and formal models provide more insight in the behaviour of a particular mutator with respect to its storage requirements. For example we have observed that the average depth of a graph is proportional to the square root of the number of cells that

the graph occupies. The experimental results are specific to the mutator and to a lesser extent application specific. The method that we have used to obtain experimental results and the analytic models are applicable to other implementation methods of programming languages.

Storage management in parallel systems is more difficult than in sequential systems because several mutators and collectors may be actively working at the same time on the same store. In principle we regard a parallel system as a collection of cooperating sequential systems. However, information about the cost of garbage collection in a sequential system, such as that described above, can not be reported directly to a parallel system. A sequential system has a store with constant access time for both the mutator and the collector. The mutator and the collector may share a single processing element or be supported by different processing elements that have access to a common store, but such a system will still be considered sequential. The main difference between a parallel and a sequential system as far as storage management is concerned is, that in a parallel system it is impossible to provide all processing elements with constant access time to a common store.

In parallel systems the mutators and collectors often work on local copies of global information and the need for information interchange between mutators and collectors is kept as low as possible. Access to local information (in a local store) is always faster than to global information (from a common store). The cost of data communication can often be kept low by copying data in order to make it locally available to a computation before starting the computation. The alternative is to perform the computation where the data resides. In both cases it is difficult to decide what to do if more than two pieces of information are required by a single computation and the information is scattered over various stores. An efficient system would keep the amount of information that must be copied small in relation to the amount of work involved in the actual computation. Most systems that copy information require a facility for referencing cells across the entire store because copied information will often contain references that should not be invalidated by moving the information around. Copying cells with references to other cells complicates the garbage collection process because the origin of a reference is no longer confined to the store where the cell itself resides. The situation is further aggravated by the fact that copied references may be copied again etc. The garbage collectors that are used in sequential systems can not be used without modification in parallel systems because of existence of copied references.

The method that we use to evaluate parallel functional programs avoids the problem of copied references by also copying the referenced cells (and recursively the cells they refer to). This will often cause more information to be copied than is strictly necessary for the evaluation of the program, but it has the advantage that sequential garbage collection methods and their analysis as described above can be applied independently to

each separate store. Garbage collection and reduction on different stores can thus be performed in parallel. Systems that copy references require individual garbage collectors to cooperate, which makes them inherently slower than ours. However, our method has two disadvantages. We sometimes copy more information than is necessary hence data communication cost may be higher. Secondly, special care has to be taken when writing parallel functional programs, in particular to avoid copying of work. The proposal of our parallel graph reduction method includes several examples of parallel functional programs and a characterisation of the programs that may benefit from parallel evaluation. We describe a method to control the data communication cost and show that this method fits in a more general frame work that can be used to aid the distribution of work over a system of parallel processors.

The functional programming language that we use is SASL,[3] because we had available some sequential applications written in this language. The parallel applications that we developed are also written in SASL. Its implementation[4] is based on combinator graph reduction. At each step the reduction process transforms the program graph by applying one of a set of predefined transformation rules (combinators) to it. Since Turner's seminal paper, refinements to the technique have been published,[5,6,7,8] which aim at improving the execution speed of combinator graph reduction. However we have deliberately chosen not to use any of these more advanced methods of implementing graph reduction because Turner's method is easier to reason about. It uses a fixed set of combinators and transforms the graph during every single reduction step. Contemporary graph reduction methods derive their improved efficiency in essence from trying to avoid the use of the graph at the expense of considerable complication, for instance in maintaining full laziness. Turner's implementation technique has the advantage that the reduction machine only knows a few dozen combinators and requires a simple algorithm to determine which transformation to apply next.

The rate at which garbage is produced by Turner's method of graph reduction is higher than with most other methods. On the average one cell is released per reduction step and reduction steps are small units of work. Typically an arithmetic operation requires a dozen reduction steps for the arguments to be put in the right place. Hence the quality of the storage manager is of crucial importance to the execution speed of the reducer. Most implementation techniques reduce the influence of the storage management system on the performance of a reducer by avoiding the use of the graph. In the current study a different point of view is taken. Among the known implementation methods Turner's method can be considered as the worst case for the storage manager (with fixed size cells). This method combines the advantages that a compact reduction system has to offer with heavy demands on the storage manager. Most problems involving the interaction between combinator graph reduction and storage management appear in their purest form. We have studied several problems in both sequential

and parallel combinator graph reduction.

1.1. Outline of the thesis

The thesis contains an introduction and five papers. In the introduction we present the principles of combinator graph reduction and describe the hybrid simulation method that we have used to experiment with combinator graph reduction. The introduction is divided in five chapters (the first is what you are reading now). The second chapter provides an introduction to graph reduction and garbage collection. Hybrid simulation is the subject of the third chapter. The fourth chapter discusses some technical issues involved in combinator graph reduction in detail, because this information is not readily available in the literature.[9,10] The fifth chapter of the introduction describes implementation details of Turner's method that influence the production of garbage. Both hybrid simulation and a thorough understanding of combinator graph reduction are expected to become more and more important as new implementation methods for sequential and parallel graph reduction emerge.

In the first three papers we report on aspects of sequential combinator graph reduction and storage management. The first paper[11] describes experiments that have been performed with a benchmark of functional programs. The execution of the programs is studied, and various execution parameters are identified and measured. An example is the average number of cells that turn into garbage per reduction step. In the second paper[12] the same benchmark is run on the same combinator graph reducer, but with different garbage collectors. A ranking among the garbage collection methods depending on the amount of available store is a result. In the third paper[13] a model is developed for one particular aspect of the interaction between combinator graph reduction and garbage collection. Based on commonly made assumptions a formula is derived for the average number of cells that turn into garbage when an arbitrary edge is deleted from a binary graph.

The last two papers describe our method of parallel combinator graph reduction. The "jobs" that are executed in parallel consist of "self contained" graphs. The number of references from one job to another is kept low and the use that is made of such non-local references is carefully controlled. When a job is created, a copy is made of the graph that represents the expression to be reduced in parallel. A strict hierarchy is enforced on the job structure. The evaluation of a parallel program starts off sequentially as a single job (the root job). A job may create a number of child jobs, which in turn may create grand child jobs etc. The only interaction that is necessary and supported takes place between parent and child. The parallel evaluation strategy has been designed such that a parent awaits completion of all its children before the parent may proceed. Due to the hierarchy of jobs thus created, garbage collection of jobs becomes

simple. The removal of the last (and only) reference to a job is explicit, while the removal of the last reference to a cell in a combinator graph is implicit and thus complicates the garbage collection within jobs. The fourth paper[14] is devoted to the development of parallel functional programs that satisfy the requirements of a hierarchical job structure. The fifth paper[15] describes the architectural context and presents experimental speed-up figures for five parallel functional programs.

Chapter 2

Functional programs and reduction: an introduction

In this chapter we provide a brief introduction to aspects of functional programming and combinator graph reduction from the point of view of storage management. For a full coverage see one of several text books.[9, 10, 16] We assume the reader to be familiar with programming in a language such as Pascal.

Programs are built from subprograms that exchange input and output. In functional programs the subprograms are functions, in non-functional programs (e.g. Pascal programs) subprograms are the functions and procedures. A subprogram may have arguments to explicitly specify its input, but input may also be taken from a global state (e.g. global variables). In non-functional programs this global state may be changed but in functional programs this is not possible. A functional subprogram applied to a particular input will always deliver the same result. This property of functional programs is called referential transparency. Most Pascal programs are not referentially transparent. It is in principle possible to write functional programs in Pascal, but this is difficult to achieve. One of the reasons is, that functions in Pascal are not "first class citizens". For example there is no way for a function to yield a function as a result.

The lambda-calculus[17] is a mathematical theory that can be used to define the notion of a computable function. The theory is equally well applicable to pure mathematical functions as functional programs and subprograms. Because the syntax of pure lambda calculus is too abstract for conveniently writing programs, the notation used for functional programming is richer than pure lambda calculus. Functional programs are in essence lambda calculus expressions with more elaborate syntax.[16] The theorems of the lambda calculus can therefore be applied to functional programs. The input to a functional program can also be expressed as a lambda expression, such that the combination of a program and its input can be viewed as a single lambda expression.

A functional program represents a computation, that can be executed by a well disciplined individual or more interestingly by a machine.[18] The lambda calculus defines the axioms and theorems that must be used to perform such a computation. The computation itself is usually referred to as reduction, since the program and the input are reduced to an answer. The answer is called the normal form of program and input. Expressions exist that do not have a normal form. For instance a program with an "endless loop" will fail to terminate such that the normal form will never be found.

If the reduction process is automated (i.e. implemented on a computer), the machine is called a reduction machine. To make the distinction between the machine that supports

the implementation and the reduction process itself, the latter is often referred to as an abstract machine. The reduction process is carried out in a series of reduction steps. Each step selects a suitable subexpression and replaces it by an answer according to a certain reduction rule. A subexpression is reducible if it matches a reduction rule known to the machine. A reduction step preserves the meaning of the program but alters the form with the effect of moving a step closer to the answer. The set of reduction rules is called the reduction system. The reduction strategy is the algorithm that, given the state of the reduction process, determines which subexpressions can be reduced next. The algorithm knows about the reduction rules that can be applied and selects only the subexpressions that match a rule. To be efficient, a machine must operate on a suitable representation of a program. The reduction system, the strategy and the representation together are called the reduction model.

To illustrate the principles of reduction we describe four versions of an abstract reduction machine that is specifically designed to calculate the factorial of an integer number. For example *fac* $3 = 3 \times 2 \times 1 = 6$. It is common practice in functional programming to denote function application by juxtaposition of the function and its argument(s). Parenthesis do not serve to delineate the arguments to functions as in most non-functional programming languages, but are used to group symbols as in arithmetic expressions. The factorial program is shown in figure (1). It uses as reduction rules subtraction, multiplication and a test for "have we performed the last multiplication yet?". While more numbers are to be multiplied, the result of the current application is the product of the current value of n and the factorial of $(n - 1)$. Hence in the definition of the factorial function we use *fac* again to recursively solve a subproblem *fac* $(n - 1)$ of the current problem.

fac $n \equiv$ if $1 = n$ then 1 else n * *fac* $(n-1)$

Figure 1 : A functional version of the factorial function

The intermediate states of the reduction of *fac* 3 are shown in figure (2). The machine necessary to reduce the expression disposes of the tables of addition and multiplication. Rules based on these tables are used in steps 5, 8, 10, 11, 14 and 15. The machine must be able to test arguments (steps 2, 6 and 12) and to decide which branch of the conditional is to be taken (steps 3, 7 and 13). Furthermore there must be a way to generate a copy of the function body of *fac* with a value substituted for the formal argument n (steps 1, 4 and 9).

```
   fac 3 →

 1.if 1 = 3 then 1 else 3*fac (3-1) →
 2.if false then 1 else 3*fac (3-1) →
 3.                      3*fac (3-1) →

 4.3*(if 1=(3-1) then 1 else (3-1)*fac (3-1-1)) →
 5.3*(if 1=2     then 1 else (3-1)*fac (3-1-1)) →
 6.3*(if false   then 1 else (3-1)*fac (3-1-1)) →
 7.3*(                       (3-1)*fac (3-1-1)) →
 8.3*(                      2    *fac (3-1-1)) →

 9.3*(2*(if 1=(3-1-1) then 1 else (3-1-1)*fac (3-1-1-1))) →
10.3*(2*(if 1=(2-1)   then 1 else (3-1-1)*fac (3-1-1-1))) →
11.3*(2*(if 1=1       then 1 else (3-1-1)*fac (3-1-1-1))) →
12.3*(2*(if true      then 1 else (3-1-1)*fac (3-1-1-1))) →
13.3*(2*(                  1                           )) →
14.3*(2                                                 ) →
15.6
```

Figure 2 : Intermediate states during the reduction of *fac* 3.

The above reduction process is not efficient, because the results of certain subexpressions are evaluated more than once. For example the subtraction (3–1) is performed three times, in steps 5, 8 and 10. This is because we have performed string reduction: for each occurrence of n in the body of the function we have substituted a copy of the actual argument expression (3–1). There are two solutions to this problem. One is to evaluate an argument expression before substituting it into the body of the function. Hence the expression is evaluated once and its value may be used as often as we like. This is called applicative order evaluation. The evaluation order as it is used in figure (2) is called normal order evaluation. The other method to improve the efficiency is to use graph reduction. Because of its importance we will discuss it in the next section.

Normal order reduction has better termination properties than applicative order reduction. It can be proved that if an expression has a normal form, then normal order reduction will find it (hence the name normal order reduction). With applicative order reduction this is not the case. For example if we reduce all (three) arguments of the conditional before deciding which branch to take the *fac* function would never terminate. Healthy reduction models guarantee that no matter what evaluation order is chosen, if we find a normal form, it will always be the right one. A healthy reduction model is one that satisfies the Church-Rosser properties, which roughly state that if there are different ways of reaching a normal form, the result will be the same. This has an important consequence: reduction steps may be performed in any order, in particular in parallel without influencing the correctness of the answer. However, care should be taken that the strategy is normalising. This property of functional programs

is the basis of all contemporary research in the area of functional programming and reduction.

An issue that we have not discussed yet is how a normal order reduction machine finds the next reducible expression (redex for short, plural redexes). We introduce different kinds of redexes by means of examples. At step 1 in figure (2) there are three real redexes and two potential redexes. A real redex can be carried out immediately, but a potential redex can not proceed until one or more of its arguments have reached a particular form. Here the real redexes are the equality test (the left-most outer-most redex), the subtraction (the left-most inner-most redex) and the application of the factorial. The multiplication is a potential redex, because we can only multiply numbers. The other potential redex is the conditional. The normal order reduction strategy by definition chooses the left-most outer-most redex. The strategy is implemented by the unwind algorithm that walks over the expression from left to right until it has recognised a function that has the required number of arguments in the right form. Some functions (multiplication) care about the form of all their arguments, others are less critical. The conditional for example requires its first argument to yield either *true* or *false*, but does not care what the *then* and *else* branches look like. All the unwind algorithm has to know is how many arguments there should be and which arguments (if any) must be normalised before a potential redex becomes a real redex. For each of these "strict" arguments in a potential redex the unwind algorithm is started from the beginning with a new goal: to normalise that argument. The current reduction process is suspended, to be resumed when the argument has been normalised. Hence the unwind algorithm operates recursively.

2.1. Graph reduction

During reduction a functional program can be represented as a string or as a graph. With string reduction arguments are copied to the places where they are needed. With graph reduction the argument itself is left undisturbed but references to it are distributed over the representation of the program. The first time that an argument value is actually needed (for instance the value of (3-1) by the equality test in our factorial program) its value is computed. The remaining references to the argument expression now refer to the value 2 that was just computed and thus share the computation. This will be illustrated with little pictures that show the graphical representation of the program being evaluated.

Before we can present the graphical form of the factorial function the notation of the operators has to be changed from infix to prefix. The conditional can be written as a function with three arguments: the test, the *then* branch and the *else* branch. The arithmetic operators are transformed into functions with two arguments. Figure (3-a) shows

the rules that change the notation for the factorial function. The result of applying the rules is shown in figure (3-b).

$$
\begin{array}{lcl}
a * b & \Rightarrow & * \; a \; b \\
a - b & \Rightarrow & - \; a \; b \\
a = b & \Rightarrow & = \; a \; b \\
\text{if } a \text{ then } b \text{ else } c & \Rightarrow & \text{if } a \; b \; c
\end{array}
$$

(a) Transformation of infix to prefix notation

$$fac \;\; n \equiv \text{if } (= \; 1 \; n) \; 1 \; (* \; n \; (fac \; (- \; n \; 1)))$$

(b) The factorial function in prefix notation

Figure 3 : Prefix notation of operators and the conditional

We draw a picture of a function application by writing the name of the function above the pictures of its argument(s). Note that this is a recursive statement: pictures of function applications can be built from pictures of function applications. To render the relationship between the function and its argument(s) obvious we draw solid arrows from the function name to each argument. The arguments themselves are drawn from left to right. Figure (4) shows the prefix definition of the factorial function in graphical form. The space saving advantage of using a graph becomes apparent if we compare the occurrences of n in figures (3-b) and (4). In the graphical form of the function body there is only one n, to which three arrows are drawn.

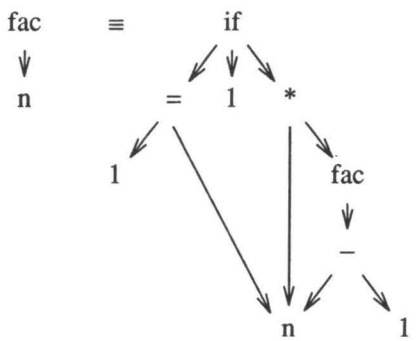
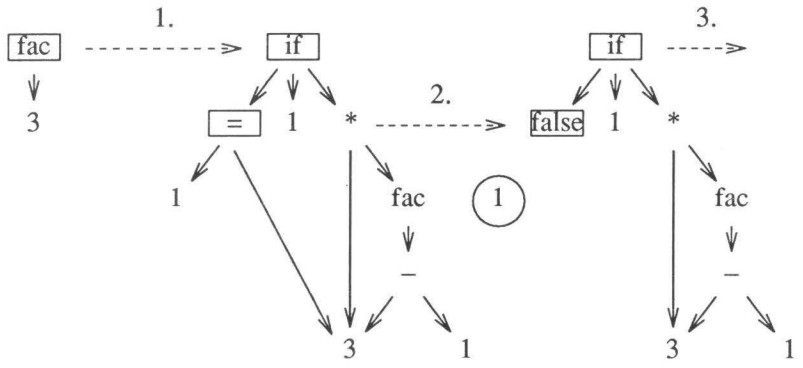


Figure 4 : Graphical form of the factorial function

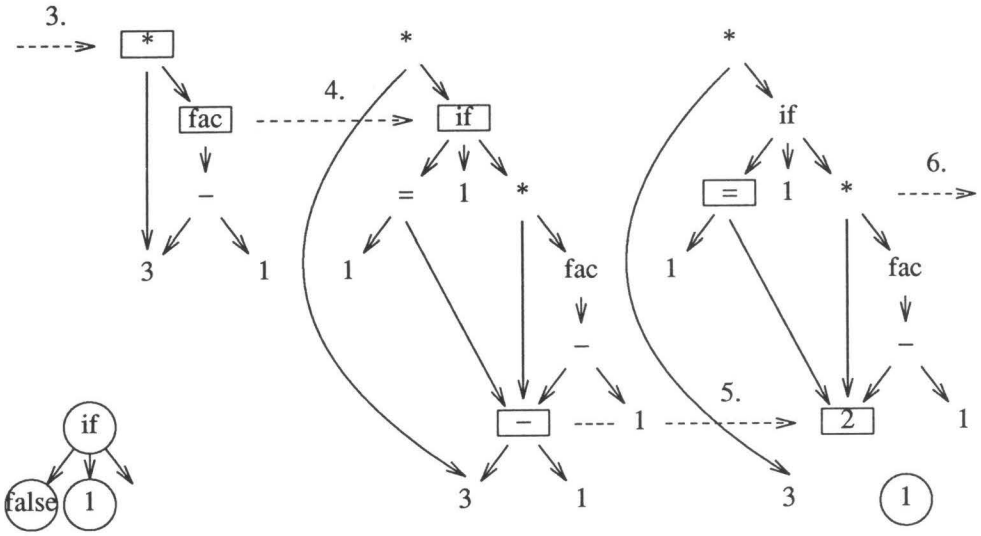
The reduction machine operating on the graphical representation of a program still has to know the rules of arithmetic, the conditional and equality test and how to instantiate

a function. Parameter substitution is now a little easier, since there is only one occurrence of the argument. Figure (5) shows the first five and the last three graph reduction steps for *fac* 3. Steps 6-9 are similar to steps 2-5. The first step makes an instance of the function body of *fac* with the argument 3 substituted for n. Note that we must copy the body, otherwise there would be no original left that we can use later to copy and substitute other argument values in. At each reduction step a dashed arrow is drawn between two boxed nodes to indicate that these should actually have been drawn as one and the same node. On paper this would make the drawings hopelessly tangled, but the machine may just destroy the old graph to make place for the new configuration. The reason we must overwrite a node is to maintain the advantages of sharing. In the fifth step for example the function application (– 3 1) is overwritten by its answer 2. Both the test and the *else* branch will benefit from this computation. We have a slight problem when a node must be overwritten by an existing node. For example all references to the node marked *if* must be redirected to the node * when the conditional selects the *else* branch in step 3. In the example we can ignore the problem because the nodes to be overwritten with an existing node are not shared, but we will come back to this issue in the next chapters. The technical term for the disconnected nodes drawn in circles is "garbage". The problems associated with managing these nodes are described in the next section and form the main topic of the thesis.

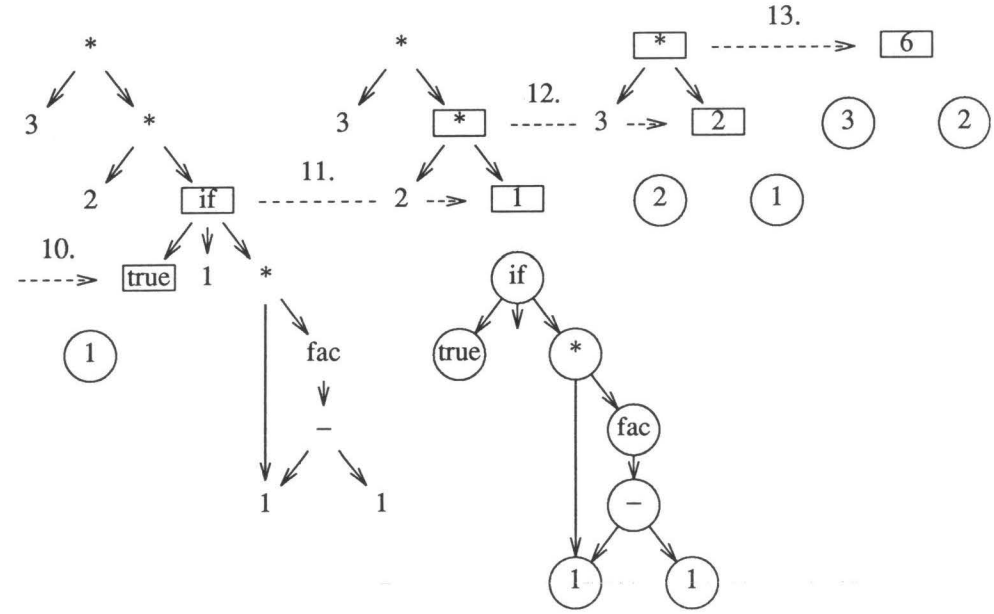
In general one must be careful with the introduction of sharing, because if a strategy is normalising for terms without sharing, it does not automatically follow that the strategy is also normalising for graphs (with sharing). However for normal order reduction it is the case that also for graphs this strategy is normalising.[19]



(a) The first two steps of *fac* 3



(b) Steps 3, 4 and 5 of *fac* 3



(c) The last three steps (steps 6-9 are similar to steps 2-5)

Figure 5 : Graph reduction of *fac* 3

Having gone through the trouble of first changing the notation of the program and secondly to draw pictures in order to calculate the factorial of 3 it is useful to compare the performance of graph and string reduction. We would like to know which method

is faster and which uses the least amount of store. The two quantities that lend themselves to being compared at this level of abstraction are reduction steps and the number of nodes required by the reduction process. More concrete figures such as the computing time in seconds can not be given because we do not have concrete implementations at our disposal. Even if we had such implementations, the comparisons would involve a host of irrelevant implementation details that are difficult to avoid.

With string as well as graph reduction we need to store an expanding and contracting representation of the program. Since computer memories are rigid in the structures they can accommodate in a natural way, a mechanism is needed, that maps the expanding and contracting structures onto the linear arrays of memory cells that computers offer for storage. If we assume that the cost of implementing such a mechanism is the same for graph and string reduction, we may use the number of nodes that must be allocated to evaluate *fac* 3 as the cost in space. The reduction steps performed with string and graph reduction of the factorial program are similar. During a reduction step the store must be accessed, but the representation (i.e. graph or string) was assumed not to influence the access cost. Hence comparing the number of reduction steps should provide a good indication of which system is faster. Making the balance we find that string reduction of *fac* 3 takes 15 reduction steps. The number of nodes required is 2 to start off with. Each instantiation of the factorial function reuses two or more nodes and requires some new nodes. All symbols except the parentheses and the *then* and *else* symbols require one node each. The first instantiation needs 9 new nodes, the second 13 and the third 17 nodes, making a total of 41 nodes. With graph reduction we need 13 steps and 2+3*7 = 23 nodes. Hence normal order graph reduction is a winner in both respects. In the summary at the end of this chapter these figures are compared with the data from the fixed combinator machine that we will introduce shortly. The difference between applicative order string reduction and normal order graph reduction is much harder to assess, because evaluation under such a radically different strategy generally requires programs to be rewritten. For instance if potentially infinite lists are used (see figure 10) indiscriminate applicative order evaluation will cause non termination. In this thesis we will not be concerned with applicative order reduction because of its restrictive termination properties.

Normal order graph reduction as it is performed in figure (5) is such an important technique that it has been given a name: lazy evaluation. Normal order reduction is lazy if it always maintains the maximum amount of sharing. The precise definition of lazy evaluation is a little more involved because it must extend to other phenomena such as the use of arbitrary data structures (see for example Field & Harrison pp. 126-130[9]). The most widely used implementation technique based on applicative order evaluation is called eager evaluation. Its precise definition may be found in the same book.

2.2. Garbage collection

Working with graphs instead of strings has the disadvantage that a garbage collector is required. This is a device that recuperates the storage that is no longer in use. A garbage collector is necessary because for instance the shared value 2 computed at the last step in figure (5-b) must remain in existence until all interested parties have used it and it is difficult to discover when the last reference to such a value will be removed. For other nodes it seems easier to detect when they are no longer needed. For instance the node 1 that is drawn with a circle around it in the third picture has never had more than one reference to it. In the next step a tree of three nodes is no longer necessary and the space occupied by all three nodes may be recuperated. The garbage collection method that we have just used is called reference counting, because by counting references to a node we decide whether it is still in use or has become garbage. There are two reasons why it is not so easy to detect when a node is no longer connected to the rest of the graph. The first is, that we must maintain a count of the number of references to the node (hence the name of the garbage collection method: reference counting). The reference count of a node is easy to derive from a picture but in a reduction machine it requires space (to store a count in each node) and time (to keep the count up to date). The second reason is, that this mechanism will not work if the graph is cyclic. For instance if one of the pointers of a node references the node itself, the count will always remain greater than zero, so the node can not be recuperated. Many applications require cyclic graphs for efficiency. Advanced reference counting methods have been proposed that under certain circumstances can handle cyclic graphs properly,[20,21] but usually at considerable cost. There are other widely used methods of garbage collection that can handle cyclic graphs. They work by elimination: starting with the node that triggered the reduction process (e.g. the node *fac* in the first picture of figure 5) mark all the nodes that can be reached by following a (solid) arrow. If no more nodes are left to mark, the remaining nodes are apparently no longer necessary and can be recuperated. The storage allocator that we use in our experiments[11] combines reference counting with mark/scan garbage collection. Nodes not involved in cycles are recuperated with the reference counting mechanism. Whenever the available storage fills up with nodes that can not be recuperated by reference counting because they are all involved in cyclic structures a mark/scan garbage collection is used to recover them. The advantages and disadvantages of various garbage collection methods is the subject of the second paper that is reproduced in the thesis.[12]

2.3. Removing bound variables

The reduction machines for the factorial problem require a sophisticated "template copying" mechanism for instantiation of functions combined with substitution of (references to) argument expressions. Although such a mechanism can be

implemented efficiently,[8] the same effect can also be achieved in a simpler way. The variables that occur in functional programs are bound variables, such as the variable n in the factorial function. The name stems from the fact that the variables are bound to a value when the function is applied. For example when we apply *fac* to 3, this value is bound to the variable n. Different instances of the same function may bind their private n to other values. The occurrences of the variable n in the function body serve as a place holder for the number of which we wish to compute the factorial. Once bound, the value of a variable will never be changed.

Because a bound variable always represents the same value, occurrences of such variables can be removed from the program text by transformation. This process is called abstraction. The idea is to mark the place where a bound variable is removed, by a special function that will put the value of the variable in the right place during reduction. These special functions are commonly called combinators because they serve to combine functions with arguments.[22] Depending on the places where the bound variables occur we use different abstraction rules. With each abstraction rule is associated a combinator that has the inverse effect of the abstraction during reduction. To remove the bound variable n from the factorial function we need three combinators S, B and C. Their abstraction and reduction rules are shown in figure (6-a) and (6-b). A reduction rule is operationally the same as a function definition; the definitions that are built into the machine are usually called the reduction rules but function definitions are derived from the application program. Abstraction is indicated by square brackets. The C abstraction rule for example is to be interpreted as follows: to abstract the variable x out of the function application $(F_x\ H)$, replace the application by one of the combinator C, and continue to abstract x out of F_x. The symbols F_x and G_x represent arbitrary expressions that depend on x. The expression H does not depend on x.

$$
\begin{array}{lcl}
[x]\ (H\ x) & \Rightarrow & H \\
[x]\ (F_x\ G_x) & \Rightarrow & S\ ([x]\ F_x)\ ([x]\ G_x) \\
[x]\ (H\ G_x) & \Rightarrow & B\ H\ ([x]\ G_x) \\
[x]\ (F_x\ H) & \Rightarrow & C\ ([x]\ F_x)\ H
\end{array}
$$

(a) Abstraction rules

$$
\begin{array}{lcl}
S\ f\ g\ x & \rightarrow & (f\ x)\ (g\ x) \\
B\ f\ g\ x & \rightarrow & f\ (g\ x) \\
C\ f\ g\ x & \rightarrow & (f\ x)\ g
\end{array}
$$

(b) Reduction rules

Figure 6 : The distribution combinators S, B and C

The S, B and C combinators are very similar. A way to remember which one is which is to observe that the S combinator di<u>S</u>tributes its third argument over the first two. The B combinator <u>B</u>inds the third argument to the second and the C combinator

Combines the third argument with the first. The abstraction rules are designed to operate on prefix notation. The appropriate representation of the factorial function has already been derived in the previous section (see figure 3-b). Figure (7) shows the abstractions necessary to remove the bound variable n. To save space abstraction on independent subexpressions are performed "in parallel". The final version of the factorial program shown in the last line no longer contains bound variables. All that remains is an expression with the combinators S, B and C and the prefix forms of the conditional, the equality test and the arithmetic operators.

```
fac  ≡
[n]( ( (if   (=1 n)) 1)   ( (*n)   (fac   ( (-n) 1)))) ⇒
(S [n]( (if  (=1 n)) 1) [n]( (*n)   (fac   ( (-n) 1)))) ⇒
(S (C [n](if (=1 n)) 1) (S [n](*n) [n](fac   ( (-n) 1)))) ⇒
(S (C (B  if [n](=1 n)) 1) (S     *   (B  fac [n]( (-n) 1)))) ⇒
(S (C (B  if   (=1  )) 1) (S     *   (B  fac (C [n](-n) 1)))) ⇒
(S (C (B  if   (=1  )) 1) (S     *   (B  fac (C     -   1))))
```

Figure 7 : Abstraction of the factorial function to the combinator version

With this result we can reduce the application *fac* 3 on our third reduction machine, which like the previous ones knows about arithmetic and conditionals. A reduction system with only a fixed and limited set of combinators such as S, B and arithmetic is called a fixed combinator reduction system. The factorial function from figure (1) is called a super combinator.[7] The difference between combinators such as S and super combinators such as *fac* is, that the latter are tailor made to the application.

The first 15 and the last 9 fixed combinator reductions generated by the application *fac* 3 are shown in figure (8). At each step only the portions of the combinator expression are shown that are currently used. These are instantiated as reduction proceeds. This is not possible with super combinator reduction since the entire structure must be searched during instantiation for occurrences of the arguments. Fixed combinator reduction is "lazier" than super combinator reduction, (see page 265 of Peyton Jones' book[10]) because it instantiates only the parts of the expression that are really necessary. Laziness implies that unnecessary work is avoided as much as possible. The benefit becomes apparent when we look at steps 9 - 15 in figure (2). The argument of the super combinator *fac* has to be substituted twice in the *else* branch of the conditional, although this branch will never be used. Since the *then* branch is small, the effect of not using it during the first $n-1$ recursive invocations of *fac* is not manifest, but it does accumulate.

```
fac ≡
   S(C(B if (=1  ) )1  ) (S *  (B fac (C -  1  ) ) )

   fac 3 →

 1.S(                  ) (                        )3 →
 2.  C(             )1 3  (                      3)  →
 3.    B if (    )3 1     (                      3)  →
 4.      if (=1 3)  1     (                      3)  →
 5.      if false   1     (                      3)  →
 6.                        S *  (                )3  →
 7.                          *3 (B fac (C -  1  )3)   →
 8.                          *3 (  fac (C -  1 3) )   →

 9.*3 (S(                   )          (           ) (C -  1 3)) →
10.*3 (  C(             )1 (C -  1 3)  ( (C -  1 3))          ) →
11.*3 (    B if (    )(C -  1 3) 1     ( (C -  1 3))          ) →
12.*3 (      if (=1 (C -  1 3))  1     ( (C -  1 3))          ) →
13.*3 (      if (=1 (  -3 1  ))  1     ( (C -  1 3))          ) →
14.*3 (      if (=1 (  2     ))  1     ( (C -  1 3))          ) →
15.*3 (      if false            1     ( (C -  1 3))          ) →
```

(a) The first 15 steps of *fac* 3

```
24.*3 (*2 (if (=1 (C -           1 (C -  1 3))) 1 (  ))) →
25.*3 (*2 (if (=1 (  -(C -  1 3) 1 ))           1 (  ))) →
26.*3 (*2 (if (=1 (  -(  -3 1) 1  ))            1 (  ))) →
27.*3 (*2 (if (=1 (  -   2     1  ))            1 (  ))) →
28.*3 (*2 (if (=1 (  1            ))            1 (  ))) →
29.*3 (*2 (if true                               1 (  ))) →
30.*3 (*2 (                                      1    )) →
31.*3 (2                                              ) →
32.6
```

(b) The last steps of *fac* 3 (the steps not shown are S, C, $-$, B, instantiate, S, C, B

Figure 8 : The normal order fixed combinator string reduction of *fac* 3

Comparison of the reduction sequences in figures (2) and (8) shows that fixed combinator reduction requires more steps than super combinator reduction (32 versus 15). The latter method inserts the arguments in the right place in a single step, whereas the former method causes the transport to be broken into a series of small steps. The small steps alone do not necessarily make fixed combinator reduction slower than super combinator reduction, since a super combinator instantiation and parameter substitution is far more complicated and time consuming than for instance an S reduction. The real reason fixed combinator reduction is slower than super combinator reduction is that during the transport of an argument intermediate place holders are required. With

string reduction an instantiation of the fixed combinator factorial needs 13 new cells and reuses the representations of *fac* 3 in the first, *fac* (*C* − 1 3) in the second and *fac* (*C* − 1 (*C* − 1 3)) in the third instantiation. The combinators *S*, *B* and *C* copy their third argument twice respectively once and once. Hence the total number of nodes required is 2 + 3*13 + 7 + 29 + 33 = 110. With graph reduction the arguments to *fac* and the 14 nodes that represent its body are shared. *S* claims two new nodes and *B* and *C* one. The total for fixed combinator graph reduction of *fac* 3 amounts to 36. Fixed combinator reduction thus requires more transient (i.e. used and almost immediately discarded) store. The gain in laziness generally does not balance this cost.

The performance of fixed combinator reduction can in principle be improved by allowing reduction steps to proceed in parallel. For example the Gross-Knuth strategy[17] repeatedly determines the set of reducible subexpressions and advances them all by a single step in parallel. This avoids the risk of run-away computations because there is at most a limited number of steps wasted on for instance discarded branches of conditionals. The Gross-Knuth strategy applied to the factorial problem performs at most two steps at the same time, as shown in figure (9). The net execution time is reduced, but the same amount of transient store is required. It is difficult to implement such small grain parallelism efficiently. In the last two papers reproduced in the thesis[14,15] we describe a method for performing coarse grain parallel reduction at a completely different level and propose a way to implement it efficiently.

```
fac ≡
   S(C(B if (=1  ) )1  ) (S *  (B fac          (C - 1  ) ) )

   fac 3 →

 1.S(                  ) (                                  )3 →
 2.  C(             )1 3  (S *  (                        )3)  →
 3.    B if (    )3 1     (  *3 (B(          ) (       )3) ) →
 4.      if (=1 3)  1     (  *3 ( (S( ) ( ))   (       3) ) ) →
 5.      if false   1     (  *3 ( (S( ) ( ))   (       3) ) ) →
 6.                       *3 ( (S( ) ( ))   (       3) )
```

Figure 9 : The first steps with parallel combinator reduction of *fac* 3

2.4. Data structures and Input/output

We have seen how function calls, arithmetic, conditionals and repetition (through recursion) are handled in a functional program. We now show that data structures and input/output can be fit into a functional framework as well. Function application can be used not only to indicate computation but also to encapsulate data. A function application normally holds its arguments together until the application is selected as a

reducible expression. The arguments are then used by the function and the application ceases to exist. If a function is supplied with an insufficient number of arguments, we have an incomplete application that can not be selected as a redex. Such incomplete applications are normally called constructors, because they provide a mechanism to structure data. The arguments to a constructor are held together as long as we wish. With data structures we also need functions to manipulate the data, since we must be able to build a structure and to take it apart. We show how that is done with an example. In most functional programming languages the list constructor *pair* (also called *cons*) is used to join two arbitrary data structures, such as numbers or other pairs. The structure may be used recursively and even made cyclic. A list of pairs is constructed using the normal reduction mechanism. For instance the program of figure (10-a) may be interpreted as the infinite list of natural numbers starting with a given value. The program is reduced in super combinator form. There is no definition for the pair function with only two arguments, hence the computation *natural* 10 terminates after one reduction step with the result *pair* 10 (*natural* (10+1)). If we define a reduction rule for *pair* with three arguments and supply a third argument to the incomplete application of *pair*, we can disassemble the data structure again. Let us interpret the third argument to *pair* as the index of the list element that we would like to select. This gives a reduction rule as shown in figure (10-b). Using these definitions for *natural* and *pair* we obtain the sequence of reduction steps shown in figure (10-c) when the second element of the list of natural numbers starting from 10 is selected. Please recall that an application such as *natural* 10 2 is to be read as "first apply the function *natural* to 10, which gives a new function that must be applied to 2".

natural from ≡ *pair from* (*natural* (*from*+1))

(a) The infinite list of natural numbers beginning with *from*

pair head tail index ≡ if 1=*index* then *head* else *tail* (*index*−1)

(b) The *pair* function with three arguments

```
   natural 10 2 →

1.pair 10 (natural (10+1)) 2 →

2.if 1 = 2 then 10 else (natural (10+1) (2-1)) →
3.if false then 10 else (natural (10+1) (2-1)) →
4.natural (10+1) (2-1) →

5.pair (10+1) (natural ((10+1)+1)) (2-1) →

6.if 1=(2-1) then (10+1) else (natural ((10+1)+1)  (2-1)) →
```

```
 7.if 1=1      then (10+1) else (natural ((10+1)+1)   (2-1)) →
 8.if true     then (10+1) else (natural ((10+1)+1)   (2-1)) →
 9.10+1 →
10.11
```

(c) Selection of the second element of the list of natural numbers from 10

Figure 10 : List construction and selection

By convention finite lists are lists whose last element is not a *pair* but a special constant *NIL*. In functional programs *NIL* terminated lists are used to structure data when the number of elements may vary. When the number of elements is fixed the *NIL* termination is unnecessary, and even if the *pair* constructor is still used, the structure as a whole is referred to as a tuple.

Because of the referential transparency requirement it is in principle impossible for a functional program to interact with its environment in a conversational way. All we can do is prepare a program and its input and send it off to the reduction machine that will reduce the combination of program and input to its answer. This mode of operation is much like "batch" processing. Real interaction with a program requires it to be able to issue prompts for data and to accept the data as it is produced. Fortunately normal order evaluation provides this possibility if we use a trick in the implementation of reduction. Suppose that a (sequential) functional program would like to read a stream of characters typed on the keyboard of a terminal. This stream can be viewed as a potentially infinite list of characters that are needed one by one. The next character from the input can be read when it is actually required by the next reduction step. Hence the input does not have to be typed in advance. Instead of loading the reduction machine with a complete program and input to be evaluated we load it with the program and a special reference to the device that will produce the input on request (e.g. the keyboard). This reference is advanced each time reduction needs the next character. Similarly the output, which would normally come out as a long list of characters that replace the program and the input, can already be displayed while the reduction machine is still evaluating the tail of the list. Neither the reference to the input device nor the early start of the display mechanism influence the referential transparency property. These so called "lazy streams" allow functional programs with normal order semantics to interact with their environment.

2.5. Summary

Functional programming offers the same expressive power as non-functional programming. Some things are easier to express in functional programs than in non-functional programs (e.g. infinite lists) and other things are more difficult to express (e.g.

interactive input and output with multiple devices). The referential transparency property of functional programs is important because of two reasons.

(1) Subprograms of a functional program can be developed and tested more in isolation from the rest of the program. The input to a function is often explicit via its arguments.

(2) Parallel evaluation of computations in a functional program can not produce an other answer than with sequential evaluation, even if the programmer had not intended the program to be evaluated on a parallel machine. Due to the use of global variables this is generally not the case with non-functional programs.

Functional programs can be evaluated in several ways. We have shown how to evaluate a simple function as a super combinator and with fixed combinators. Super combinators generally allow for a faster implementation, but require a sophisticated instantiation mechanism. A fixed combinator machine is less complicated to implement and reason about. Hence, it is more suitable as a basis for experiments with storage management. In the implementation of both methods strings or graphs can be used to represent the expressions being evaluated. With normal order evaluation, graph reduction gives better results than string reduction. Table (1) summarises the figures for *fac* 3 on the four versions of the factorial reduction machine. The percentage fewer nodes or reduction steps is shown as the gain of graph over string reduction.

	Nodes claimed			Reduction steps		
	string	graph	gain	string	graph	gain
super combinators	41	23	44%	15	13	14%
fixed combinators	110	36	67%	32	25	22%

Table 1 : Characteristics of *fac* 3 with different implementation methods

Chapter 3

Hybrid simulation

The subject of this chapter is hybrid simulation. This is the experimentation technique that we have used to collect information on the behaviour of combinator graph reduction. In our hybrid simulation a system is implemented partly by prototyping, partly by simulation. A system model defines the interfaces and interaction between prototyped and simulated components. A full simulation gives precise and detailed information on the simulated object, but the investment both in terms of man power and computing power to perform full simulations is enormous. The cost would by far exceed the resources available to a small university research group and we would not have been able to fully simulate anything more than a system running toy applications. However a more crude estimate is often just as appropriate, in particular when a system is still in its initial design phase. Many facets can then safely be characterised by "an order of magnitude". Hybrid simulation gives such possibilities. Hybrid simulation allows for selected system components to be replaced by prototype implementations, which sometimes need not even be efficient. These components operate in real time rather than simulated time. Hybrid simulation requires a reliable model of the system being simulated. This model serves as the "glue" that holds the simulated or prototyped components of a system together.

In the remainder of this chapter we give two examples of hybrid simulations. The first example describes an experiment that was designed to investigate the effect of a key parameter on the design of a parallel reduction machine. The second example gives a full account of the hybrid simulation of job based parallel graph reduction.

3.1. A small experiment

The following example serves to support the claim, that sometimes it is more efficient to actually build parts of a system and perform measurements, than it is to write a simulator. A key issue in the design of parallel architecture is the placement and the accessibility of the store. The question that we would like to answer is, whether providing each processing element with access to a global store inevitably introduces a bottleneck.

An experiment has been performed to count the number of memory accesses generated by a program that copies binary trees. It is assumed, that this activity has to be supported efficiently for graph reduction. The experimental setup includes a processor with on board memory and a separate memory array. The program, its data and stack

are stored on board, whereas the heap space is allocated all by itself in the separate memory array. The two components are connected via a high speed bus.

The test program first generates a binary tree with 1596 nodes and then copies the tree to a contiguous area of store. The tree copying routine was written in assembler and is fully optimised. The total time required for copying the tree was measured using a logic state analyser connected to the memory array. We found that copying one node requires on the average 46 μ seconds. One node occupies 8 (16-bit) words, which is the largest data item that can be transferred in a single access. This means, that on the average the heap is accessed once every $46/(2\times8) \approx 3$ μ seconds. However, the particular memory array, allows for a much higher access rate; the maximum access times are 210 nano seconds for a read operation and 320 nano seconds for a write operation.

The conclusion is, that the memory array containing the heap is heavily under-utilised by the particular test program. The processor accesses the heap only 10% of the time. During the rest of the time it performs computations or accesses the on board memory to fetch instructions etc. This experiment shows, that a globally accessible store will become a bottleneck only if about 10 processors are allowed to access it simultaneously. This model ignores the arbitration that is necessary to regulate access to a memory that is accessible to several processors, but it could be refined to include arbitration effects. The figure that we have derived gives us an initial estimate on which further work can be based.

3.2. Hybrid simulation of parallel graph reduction

The experimental results used in this thesis have been obtained from a software package that partly simulates, partly emulates a parallel reduction machine. A sequential combinator reducer according to Turner's method is at the heart of the system. It allows for extensive statistics gathering on various aspects of graph reduction and storage allocation. The parallel reduction strategy is simulated by a synchronous system of communicating concurrent processes. A process consists of a sequential reducer, a storage allocator, a set of communication channels and a fair amount of store (several Mbytes). A communication channel supports one way communication between two processes or between a process and a filing system. The combinator code that results from compilation of a functional application program is supposed to be available on the filing system before the simulation is started. The code consists of a series of (compiled) function definitions and an expression that represents the main computation. Once it is created, a reducer process reads the function definitions from the filing system. Initially one reducer process, the "root" reducer is present. It is supplied with the main expression to normalise. A reducer process may create instances of itself, to normalise newly created parallel jobs. A pair of communication channels

serves to pass the representation of a job from parent to child and to return the representation of the normal form as soon as it is available. The system is synchronous, since reduction of a parent job must wait until the child has completed its job. The normal form produced by the root process is stored on the filing system. Processes not only have direct access to the filing system to read the function definitions, but also to store diagnostic output and simulation data.

3.2.1. Mapping of jobs onto Processes

An issue that is difficult to resolve in a real (as opposed to simulated) implementation of a parallel machine is the mapping of parallel jobs onto the available processing elements. We will show that a simulated approach allows for a simple mapping strategy that can be guaranteed never to exceed the machine capacity, while still providing the required experimental data. The price to pay is an increase of simulation time.

The cooperation between reducer processes as part of the parallel simulation is perhaps best explained using a representative application program. We have chosen the quick sort algorithm to serve this purpose. Figure (11) shows the algorithm programmed in SASL.[3] The program consists of a main application *QuickSort* $(3, \cdots, 7)$ and a series of local function definitions. All definitions with the same name and indentation that appear under a *WHERE* belong together. For instance there is one definition of *QuickSort*, with two alternatives selected by pattern matching on the argument. If the list to be sorted is empty, which is written as (), the empty list is produced as the answer. In the second alternative (for a non-empty list) the function body of *QuickSort* is formed by a three way conditional. The arrow ($\rightarrow$) connecting a condition and a clause should be read as *then*. The third (*else*) clause *QuickSort* m ++ (a : *QuickSort* n) applies when the first two conditions both yield false. In the annotated quick sort algorithm jobs are created as long as the length of both sublists to be sorted exceeds a certain threshold (2 in the example). In the first *then* clause the recursive quick sort of the sublists is performed as two separate jobs, one to apply *QuickSort* to the left sublist m and the other to normalise (*QuickSort* n). If both sublists are too short the algorithm switches to a strictly sequential version (*QuickSort'* in the second *then* clause). The split function of the latter is not shown in the figure (it is identical to that of the parallel quick sort). If one of the sublists is too short the sorting process continues in a sequential fashion, but the longer sublist may still be sorted in parallel later if its pivot is more suitable (*else* clause). The symbol ++ denotes the infix operator that appends the right list argument to the left.

```
Threshold          = 2
QuickSort (3, 0, 11, 10, 13, 17, 14, 12, 18, 16, 15, 1, 8, 9, 6, 2, 4, 5, 7)
WHERE
QuickSort ()       = ()
QuickSort (a :x)   = (lm > Threshold) & (ln > Threshold)
                       → sandwich append ((QuickSort , m) , (QuickSort , n))
                     (lm ≤ Threshold) & (ln ≤ Threshold)
                       → QuickSort' m ++ (a :QuickSort' n)
                     QuickSort m ++ (a :QuickSort n)
                     WHERE
                     m ,lm ,n ,ln                =  Split a x () 0 () 0
                     append left right           =  left ++ (a :right)
                     Split a () m lm n ln        =  m ,lm ,n ,ln
                     Split a (b :x) m lm n ln    =  b < a
                                                      → Split a x (b :m) (lm+1) n ln
                                                    Split a x m lm (b :n) (ln+1)
                     QuickSort' ()               =  ()
                     QuickSort' (a :x)           =  QuickSort' m ++ a :QuickSort' n
                                                    WHERE
                                                    m ,n    =    Split' a x () ()
                                                    ...
```

Figure 11 : Quick sort application

The *sandwich* function is a program annotation that tells the reducer to generate jobs. The annotation is implemented as a special primitive instruction (a combinator). The associated rewrite rule as used in the quick sort problem is shown in figure (12), the general case is discussed in the fourth paper that is reproduced in the thesis.[14] The *sandwich* combinator first converts the list structures ((*QuickSort* , *m*), (*QuickSort* , *n*)) etc. into applications (*QuickSort m*) and (*QuickSort n*). It then turns both applications into jobs, which are reduced to normal form in parallel. Upon completion of the jobs, the function *append* is applied to its arguments (the sorted sublists) in the normal (lazy) fashion.

```
sandwich  append  ((QuickSort , m) , (QuickSort , n))
                                  →append  (QuickSort  m)  (QuickSort  n)
```

Figure 12 : Sandwich combinator as used for the quick sort problem

When the quick sort program is executed it develops a job tree that depends on the

data items to be sorted. The structure emerging from the given input data is shown in figure (13). The pivot values of the data are shown in the circles and the parts of the lists to be sorted in parallel as left and right branches. Please note, that the split algorithm of figure (11) reverses the sublists.

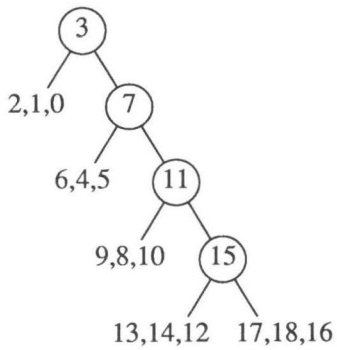


Figure 13 : Canonical job structure

The sandwich annotation in jobs causes new jobs to be generated, but before a job splits, it generally performs some reduction steps itself. This part of the reduction is called the fork job. Similarly, after the results of the child jobs have been collected, some computation must be performed to merge the results (the join job). A job that does not cause further generation of child jobs is called a mid job. Figure (14) shows the job tree for the quick sort problem, with the number of reduction steps (in roman font) required for each mid job, fork job or join job. The figures in underlined roman font represent the number of nodes transported as a result of spawning jobs or returning results. The diagram has to be read in the same way as a road map. For instance starting at the top and travelling downwards, there are 2044 reduction steps (a fork job) to be performed before we arrive at the first bifurcation point. Two jobs are created at this point, one to sort the smaller list (2, 1, 0) and the other to sort the remaining numbers (7, $\cdots$, 11). Turning right we transport the 481 nodes that represent the smaller sublist. This mid job requires 701 reduction steps to produce a sorted list consisting of 7 nodes (3 pair nodes, *NIL* and three data values). Here the similarity with the road map ends, because in the diagram there is a discontinuity that is equivalent to the time that we have to wait until the second job has been reduced as well (dashed line). At the bottom of the diagram the small sorted sublist is returned to the parent. The latter needs 12 reduction steps to merge the small list and the remainder (31 nodes) that is sorted in the other branch. The merge phase has been drawn with a scale 70 times larger than the split phase.

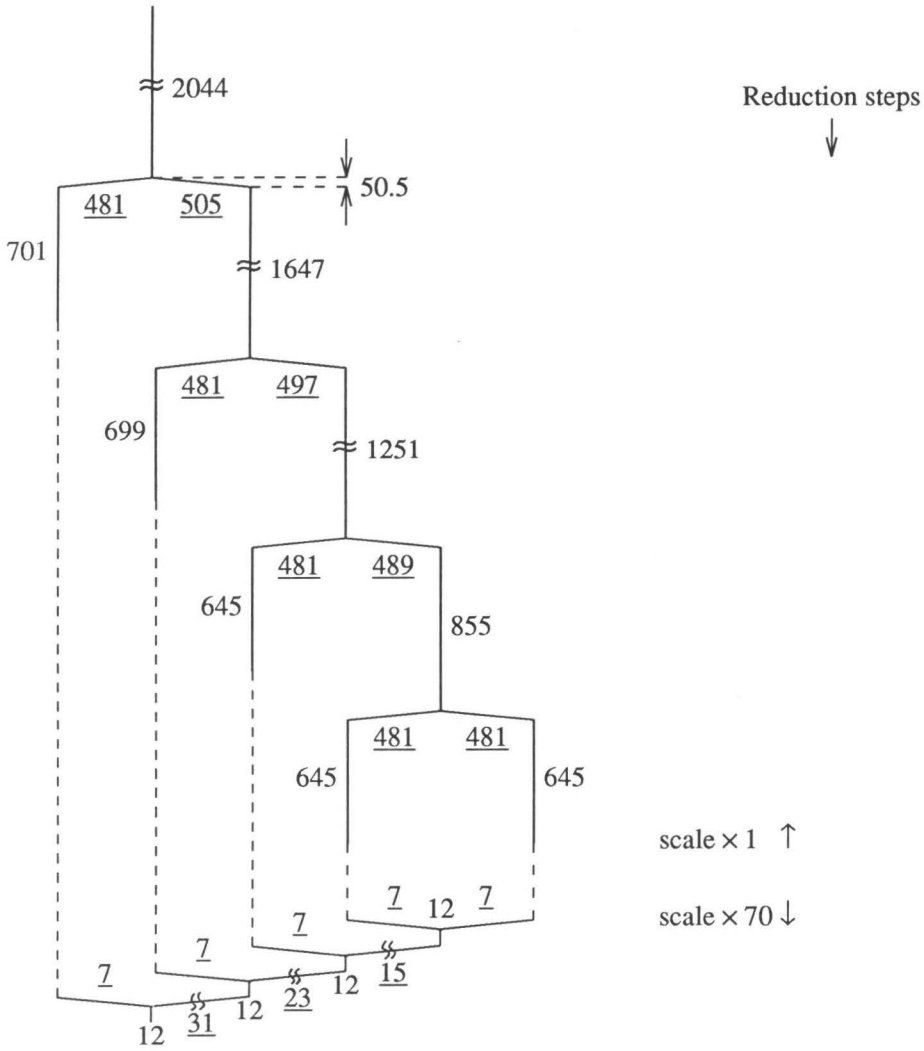


Figure 14 :Job structure with reduction steps and transport costs ($T = 10$)
Reduction steps are in roman font, transportation cost in underlined roman.

To some extent the diagram reflects reality, in that both reduction and data transport are shown to take time. The number of nodes that can be transported in unit time enters the simulation as a parameter T. The average time per reduction step is used as the unit of time. The value used for T to draw the diagram is 10 nodes/step. Modelling a complex data communication network by a single parameter is a gross simplification, that can be justified only because the results sought are a first order approximation. It allows crude estimates to be made of the system performance, which can be refined later in more detailed simulations.

The most important result that may be calculated from the data shown in figure (14) is the speed up in computation that may be achieved with the given program and input data. If the transportation costs are ignored ($T = \infty$), we find that the parallel computation requires 6490 (= 2044 + 1647 + 1251 + 855 + 645 + 4 * 12) reduction steps. Had the same computation been performed sequentially, then 9180 (= 6490 + 701 + 699 + 2 * 645) reduction steps would have been required. The best possible speed up factor for this problem is therefore 9180 / 6490 = 1.41. If transportation costs are included, the speed up factor is a little lower. Suppose that $T = 10$ as shown and that two child jobs of the same parent can be transported in parallel. The time required to evaluate all the jobs is now 6695 (= 6490 + 50.5 + 49.7 + 48.9 + 48.1 + 0.7 + 1.5 + 2.3 + 3.1) reduction steps. Hence the speed up factor with data communication cost accounted for is now 9180 / 6695 = 1.37. From the diagram (which has been drawn to scale), we can see that the maximum number of jobs that can be evaluated parallel with the given input is 2.

The parameter T is a configuration constant, which may be determined experimentally. For the configuration that we have built it is about 200, but with a fast sequential reducer instead of the current one that is slowed down by the massive amount of statistics gathering, it is more likely to be near 1. The current reducer executes 50 fixed combinator reductions per second on a VAX-11/750.†

3.2.2. Linear string of processes

To obtain the figures necessary for performance evaluation, there is no need to actually perform the quick sorts in parallel. It is sufficient to traverse the job tree in pre-order, such that all child jobs to a certain parent job are performed in order: the parent awaits completion of a child job before starting the next child. The parent itself proceeds as soon as the last child job has terminated. Figure (15) shows the job structure of figure (14) mapped onto a linear string of processes. The main expression is evaluated by process 1, both its children (the mid job requiring 701 steps and the fork/join job requiring 1674+12 steps), one by one on process 2 etc. The value 10 for the parameter T has been incorporated in the scaling of the diagram, such that the correspondence between the real time axis and the activity is exact.

† VAX is a trademark of Digital Equipment Corporation

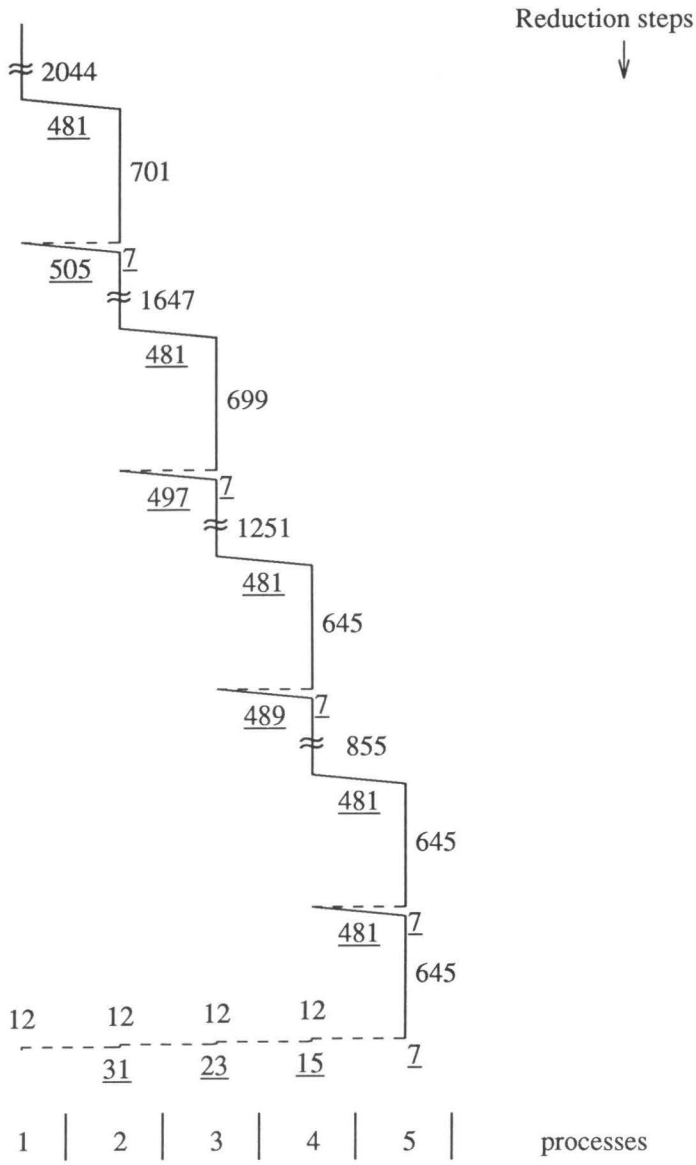


Figure 15 :Mapped job structure ($T = 10$)
Reduction steps are in roman font, transportation cost in underlined roman.

The maximum depth of the job tree generated by an application determines the number of processes required to perform the simulation. Although the application program is a parallel one, the simulation itself is not parallel at all. One could argue, that instead of creating concurrent processes of which only one is allowed to execute, it would have been better to keep the simulation in a single process. However the organisation such

as we have developed it allows for real data communication performance between neighbours to be measured. The job and result graphs that would appear in the real parallel machine are available in the simulation. We can measure how long it takes to physically transport the graphs through a communication channel. That it takes long to arrive at the point where a particular job or result is created does not influence the accuracy of these measurements. In the next section we discuss further advantages of the hybrid simulation method.

On the hybrid simulator we have compared the performance of two different implementations of the algorithm that we use to send a graph through a communication channel.[15] One version of the algorithm was hand coded in Motorola MC68010 assembler,[23] the other written in Modula-2.[24] Both algorithms maintain a stack of nodes explicitly and do not use procedure calls. The assembler version makes extensive use of the features of the processor, such as "move multiple" instructions and "auto increment" and "auto decrement" addressing modes. The performance gain was found to be a factor five of assembler over Modula-2. The transport rate of the assembler version exceeds 10000 nodes per second.

3.2.3. Load balancing

The major difference between the simulation package and a real parallel machine is the lack of a dynamic load distribution mechanism. With the fixed mapping of jobs onto processes as it is described above the need for such a mechanism has vanished. This is an advantage, because it allows for a simple implementation. Furthermore it guarantees a strict separation of two important issues: controlled generation of parallel jobs on the one hand and the mapping of jobs onto processors on the other. The current implementation allows for any number of jobs to be evaluated in parallel, because they are actually evaluated sequentially. Each application of the sandwich combinator that fires two or more jobs is scheduled such, that the jobs reduce to normal form one by one. The parent process is forced to await completion of all its children before it may proceed.

Performance estimates for the real machine can not be made unless the influence of load balancing is incorporated in the simulation. On a real machine one would generally create more jobs than there are processes, to have work available when a job terminates. With the current simulation we are able to derive approximate results that apply to the real machine, because we have all the scheduling information available: the sizes of job and result graphs, the duration of fork, mid and join jobs and the precedence relations between jobs. For example it is not difficult to reconstruct the canonical job tree shown in figure (14) from the data and precedence relations in figure (15). If insufficient processes are available to absorb the entire population of jobs at a

particular instance a decision must be made to evaluate certain jobs before other jobs. Once a set of scheduling data has been derived from a simulation run we may independently experiment with scheduling policies. So far we have used an optimal scheduling policy[15] and we are currently investigating more realistic policies.[25]

3.2.4. Implementation

The linear string of processes has been implemented in two different ways. On our UNIX† host computer a parent process constructs two pipes and forks a child process, which in turn constructs two pipes and forks etc. To reduce start up cost (i.e. the reading of the code for the user defined functions) we have made the process persistent. This means that once started, a process accepts a series of jobs from the parent until the parent explicitly tells the child process to terminate. The second implementation uses a five processor Motorola 68010 system, where the processors are arranged as a linear string. Here each physical processor supports one reducer process, which accepts a series of jobs as described above. Figure (16) shows the configuration of the first three processors. The root processor (shown to the left) is connected to the host via Ethernet. Each processor is connected to the host via a 9600 Baud terminal line. These are used for bootstrapping and diagnostic purposes as well the connection to the filing system. The communication channels between the processors are implemented with two buffers that share a dual ported memory and four interrupt wires. The data to be transported is stored in a buffer, which is accessible to both neighbours. One neighbour signals the availability of a packet by sending a signal along its *full* interrupt wire. After taking the data out of the buffer, the recipient acknowledges the signal by sending an *empty* signal over the return wire. Transport in the opposite direction uses the other buffer and the remaining two interrupt wires.

† UNIX is a trademark of Bell laboratories

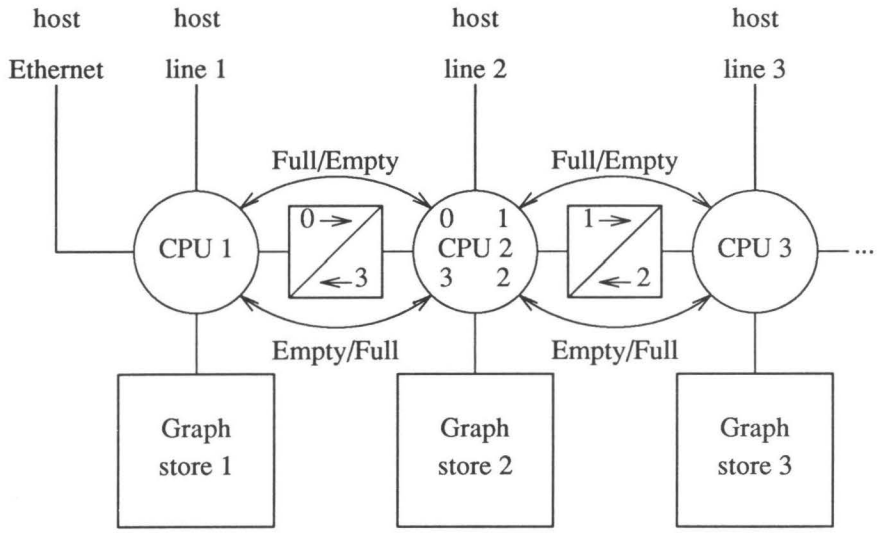


Figure 16 : Network topology

Figure (17) shows the configuration of one complete processor system crate and part of its right neighbour. Each processor has access to 256 Kbytes on board memory, 128 Kbytes of dual ported memory and 4 Mbytes of bulk memory. The code for the reducer and its supporting operating system are loaded in the on board memory when the system is bootstrapped. Three Mbyte of bulk memory are reserved for the graph and one for the recursion stack of the reducer.

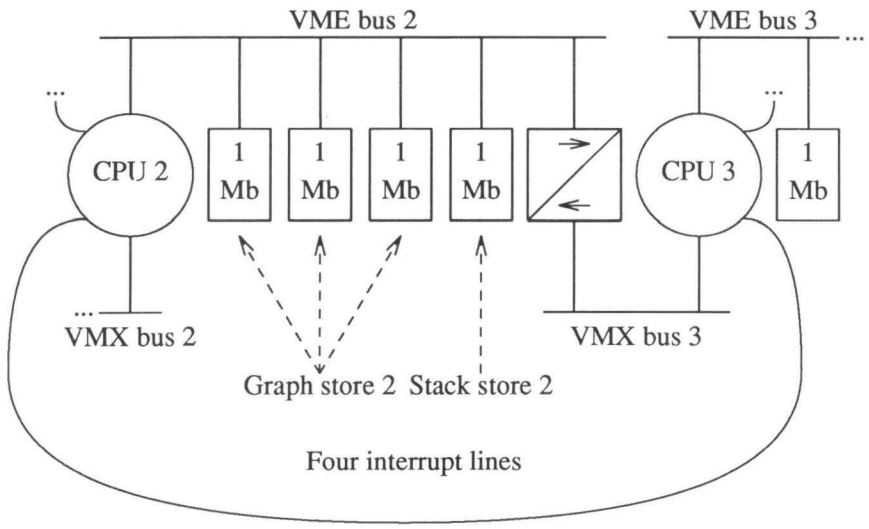


Figure 17 : System configuration

3.2.5. Software structure

The software components of the simulation package are schematically represented in figure (18). The components are used in the following way to run a SASL program on the machine. First a SASL program is compiled into combinator code. Then the system is bootstrapped (with the help of the ROM monitor) to load all processors with a copy of the reducer. Programs are available to bootstrap via Ethernet or via the terminal line(s). Each reducer reads simulation options (e.g. what statistics are to be gathered) from the terminal line that connects it with the host. The reducers then perform initialisation, which includes reading the compiled function definitions. Finally the root reducer reads the combinator code that represents the main expression from the filing system and reduction may begin. Output is produced using the terminal lines connected to the reducers. The scheduling data is further processed by a separate set of programs to calculate performance figures from the raw data. Similarly a suite of programs serve to process the statistical data that is gathered. Off line data processing takes place on the host.

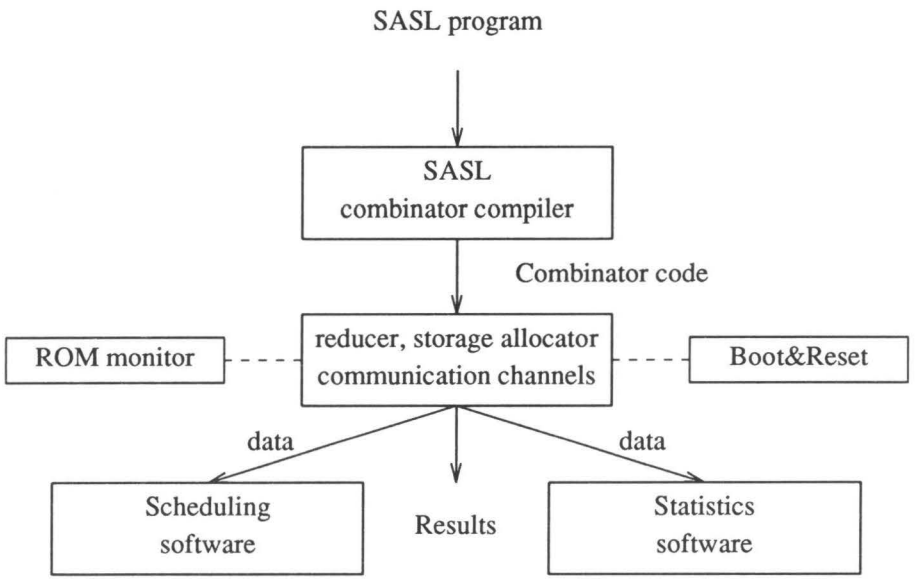


Figure 18 : Software overview

Even though the implementation of the hybrid simulation system was claimed to be simple, the amount of work involved exceeded a man year. Table (2) presents a summary of the effort that was necessary to develop the software. Each combination of a definition and an implementation module is counted as one module. Blank lines and comments are not included in the line counts. The reason the reducer contains a relatively large percentage of assembler is because it contains several garbage collection algorithms written in assembler.[12] The average number of lines per Modula-2 procedure is 24448/1703 = 21.2. Our benchmark of functional programs[11] shows an average of 2308/380 = 6.7 lines per SASL function, with the same method of counting lines. Functional programming apparently causes functions to be smaller than imperative programming.

	Operating system	Reducer and storage manager	Test programs	Total
Modula-2 source modules	36	65	29	130
Modula-2 procedures	273	748	128	1149
Modula-2 source lines	5736	15038	3674	24448
assembler source lines	132	1555	16	1703
Man months	3	9	1	13

Table 2 : Development of the hybrid simulation software

3.3. Conclusions

Hybrid simulation is a useful method to investigate the consequences of certain design considerations. It is a compromise between the construction of a full simulation and a complete prototype. The accuracy of the results depends to a large extent on the quality of the system model that underlies the hybrid simulation. The method has been used to investigate the following design issues:

- With the (commercially available) hardware components that we are using, a global store will become a bottleneck if at least 10 processors are trying to access the store at the same time.
- Parallel graph reduction based on jobs allows for a performance gain with respect to sequential graph reduction.

In the next chapters we will encounter several other experiments and results that have been obtained by hybrid simulation.

Chapter 4

Turner's method of combinator graph reduction

In this chapter we describe some important aspects of Turner's method of combinator graph reduction.[3,26,4] The emphasis is on the interaction between reduction and storage management. In general a combinator in Turner's model does not claim more than a few new storage cells. We exploit this property for instance to tell the storage manager before a combinator is fired how many cells will be necessary. An important exception to this rule is formed by the combinators that support pattern matching. We therefore describe in considerable detail how pattern matching is implemented. Since some garbage collection methods have difficulties in handling cycles efficiently we also study the creation of cycles in the representation of combinator expressions. In the next chapter we continue the discussion of Turner's method with a description of various optimisations to its implementation.

A SASL program consists of a series of global function definitions, and a single function application (the main application). A global function definition may contain local function definitions as *WHERE* expressions. The program is compiled into combinatory code before it can be executed. The compilation process takes each function definition and the main application in turn and removes the bound variables using a bracket abstraction algorithm. What results is a list of named combinations and the main combination. At this point, function names may still appear in the combinations as free variables and the individual combinations have a tree structure. A linkage process replaces the function names by references to the appropriate combinator expressions. This turns the structure into a graph; the linkage of combinations may introduce both sharing and cycles. If after linkage free variables are still present, an error message results. The actual reduction process rewrites the graph in a number of reduction steps until a normal form or an error results.

The syntax of a combination is given in figure (19). The names of the user defined functions appear as *identifiers* in the combinator code. The *combinators* have reserved names, such that they can not be used as identifiers. A *character* has to be preceded by a percent sign (%) and a string is represented as a list of characters such as in (%A:%B:%C:NIL).

combination:	term | combination term
term:	"(" list ")" | combinator | identifier |
	"%" character | number | "TRUE" | "FALSE" | "NIL" | "FAIL"
list:	combination | combination ":" list

Figure 19 : Syntax of combinator expressions

4.1. Informal operational semantics of a combinator expression

Combinators control the evaluation of the program via rewriting of the graph that represents the program. There are two categories of combinators: those that perform computations and those involved in housekeeping. The combinators that perform computations are strict in at least one argument. They are called strict combinators (even if they are non-strict in other arguments). Examples of strict combinators are the arithmetic and logic operators and type testing functions. The remaining (housekeeping) combinators are called pure combinators. These combinators serve to transport arguments to the right place. With data values, such as 1 or *TRUE* no rewrite rule is associated. Such constants do not require arguments and can not act as functions.

When supplied with the required arguments a combinator may be "fired". The rewrite rule associated with the combinator is then applied to the expression in which it appeared. The strict combinators, since they have particular requirements about the form of their arguments, normally cause other reductions to take place before their own rewrite can be performed. The pure combinators are indifferent with respect to the form of their arguments and perform the rewrite immediately.

4.2. Standard pure combinators

The standard pure combinators are S, K, I, their optimisations, the uncurry combinator U and the fixed point combinator Y. The rewrite rules associated with these combinators are shown in figure (20). The colon (:) represents the infix list constructor; the element to the left forms the head of the list and the element to the right the tail.[11] It is explicitly indicated when lists are supposed to be terminated by *NIL*.

B f g x	→f (g x)
B' k f g x	→k f (g x)
Bp f g x	→f : (g x)
C f g x	→(f x) g
C' k f g x	→k (f x) g
Cp f g x	→(f x) : g
I a	→a
K a b	→a
S f g x	→(f x) (g x)
S' k f g x	→k (f x) (g x)
Sp f g x	→(f x) : (g x)
U f a	→f (HD a) (TL a)
Y f	→f (Y f)

Figure 20 : Standard pure combinators

The notation used to express the rewrite rules is inadequate in the sense that sharing, which makes graph reduction so attractive, is not clearly expressed. For instance in the right hand side of the rule for S (in figure 20) both occurrences of x should be interpreted as referring to the same subexpression, as is shown in figure (21). The last action performed by a rewrite rule in graph reduction is to overwrite root of the subgraph that represented the reducible expression. In figure (21) this is schematically represented by the dashed arrow that connects the (boxed) roots. If the result is a new application, the left and right pointers of the root are redirected to the appropriate subgraphs. If the result is a scalar, its value can be stored directly in the root node. Rewrite rules, such as the K rule, but also special cases of various strict combinators must deliver an existing node (i.c. an argument) as the result. This can be implemented by turning the root of the subgraph into an indirection node, that threads the references to the root further to the result. Indirections are elided when they are encountered during the unwind and rewind of the left spine. Hence if the root is not shared, the indirection disappears immediately during the next rewind.

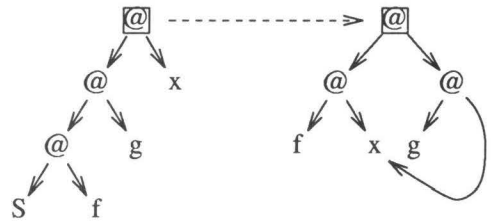
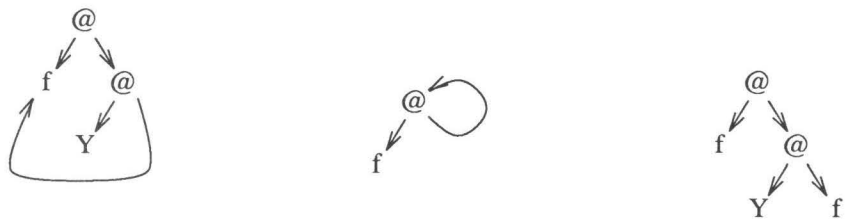


Figure 21 : Graph rewrite rule for the S combinator

4.2.1. Variants of the Y-rewrite rule

For the fix point combinator Y there are three possible ways to interpret the rewrite rule. A cyclic graph is chosen as result when the combinator is applied (figure 22-b). A non-cyclic evaluation strategy for the Y-combinator is shown in figure (22-a). With string reduction the argument f is copied (figure 22-c).



(a) non-cyclic graph reduction (b) cyclic graph reduction (c) string reduction

Figure 22 : Three possible ways of performing the Y reduction

The first and the second method cause the argument f to be shared. The second method yields a graph normal form in one reduction step, whereas the first method adds a new node to the graph, each time the Y-combinator is fired. The first method has the advantage, that no cycles are created in the graph. Although this is not the only source of cycles in SASL programs (the other one being the linkage mechanism) avoiding this particular one brings some advantages to the storage allocator. In the next chapter and in the first paper reproduced in the thesis[11] the performance under the different interpretations of the rewrite rule for the Y-combinator are compared.

The non-cyclic Y reduction has the disadvantage, that it does not maintain full laziness if $(Y\ f)$ represents a list and components of the list are used both by f and by the expression in which $(Y\ f)$ is embedded. The non-cyclic reduction will cause

recomputation of the list or parts thereof, even though f is always shared, either directly or indirectly. The sharing in the cyclic case precludes recalculation. The next section presents a detailed example of the differences between using the cyclic and non-cyclic interpretation of the Y-combinator.

4.2.2. Multiple *WHERE* clauses

An important application of the Y-combinator is in the handling of multiple *WHERE* expressions. In SASL there is no distinction between a *WHERE* clause with recursive definitions and one without such definitions. In for instance LML[8] *LETREC* and *LET* distinguish between recursive and non-recursive local function definitions (which in LML precede the function definition to which they apply). *WHERE* in SASL is always interpreted as a "*WHEREC*". During the compilation of SASL into combinatory code, the parameters of all definitions that occur under the same *WHERE* are abstracted out of their function bodies. The group of function bodies is then combined (in the order of occurrence) into a single tuple. The names of the functions are tupled in the same order as the bodies. The name pattern thus constructed is abstracted out of the tuple of function bodies. Even if the *WHERE* expressions are not recursive or mutually recursive, the tupling will make the definition as a whole recursive if at least one of the *WHERE* definitions is used in another one. An example of a function with two *WHERE* clauses and its combinatory code is shown in figure (23). Here the tuple of function names becomes $(g\ ,\ h)$ and the tuple of function bodies $(1-h\ ,\ 10)$. In the example h occurs both in the name and the body tuples. This is usually the case, hence the Y combinator is almost always necessary to make the individual *WHERE* expressions available to the group. If the *WHERE* definitions are totally unrelated, the combinatory code to uncurry the individual clauses for use within the *WHERE* tuple is wasted. The combinator expression thus generated for the entire group of *WHERE*-definitions will have the form $Y(U(U\ldots)$, with one fewer application of U than there are definitions at the same level under the *WHERE*. In the combinator code for the definition to which the *WHERE* clause applies, applications of U are present to extract the appropriate definitions for use there.

global definition	$f = 1 + g$	WHERE
where clause 1		$g = 1 - h$
where clause 2		$h = 10$
compiled code	U (K (PLUS 1))	(Y (U (B K (Bp 10 (MINUS 1)))))

Figure 23 : A non-recursive *WHERE* definition

The example program executes two reduction steps before the Y-combinator is fired. These steps are shown in graphical form in figure (24). The U-combinator prepares the code to select the code for g and h out of the *WHERE* tuple via applications of *TL* respectively *HD*. Since h is not used by f, the K reduction step removes the selection of its code. The code for the body of the *WHERE* tuple: (*B K* (*Bp* 10 (*MINUS* 1))) is not shown in the graphs (ellipses).

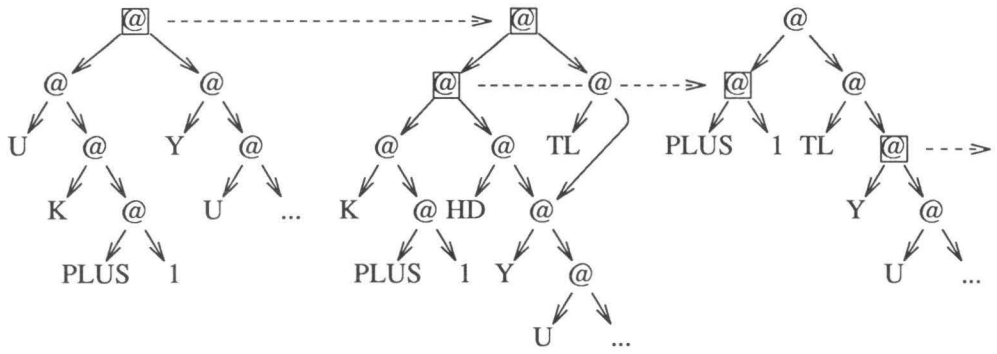
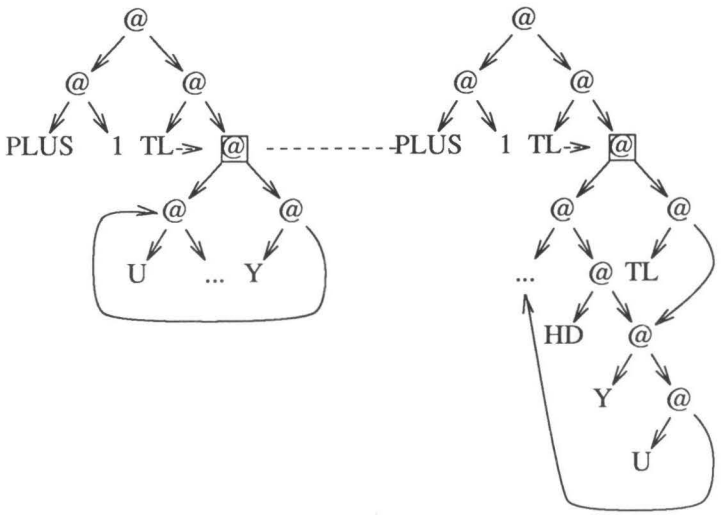
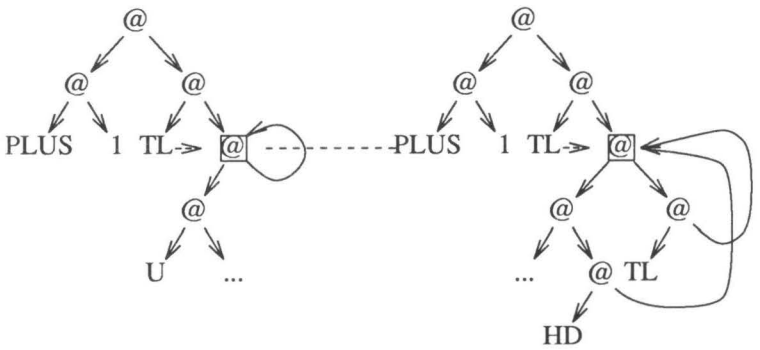


Figure 24 : The first two reduction steps (U and K)

The next combinator to be executed is the Y that precedes the *WHERE* tuple. Depending on the particular interpretation of the Y rewrite we arrive at the left most graph of figure (25-a) or (25-b). The *TL* combinator requires its argument to be a tuple, such that the U combinator is fired. This prepares the selection of the code for g and h for use within the *WHERE* tuple. The resulting configurations under both interpretations of the Y rule are shown as the right most graphs in figure (25-a/b).



(a) Non-cyclic Y reduction (steps Y and U)



(b) Cyclic Y reduction (steps Y and U)

Figure 25 : The next two reduction steps (Y and U)

Evaluation of the *WHERE* tuple proceeds with the B, K and Bp reductions (not shown). The argument of (the oldest) *TL* has by then become a tuple, with 10 as head and *MINUS* 1 (*HD* $\cdots$) as tail. The application of *HD* is still the one shown in figure (25). The cyclic interpretation of the Y rule thus yields an application of *HD* to a tuple cell, whereas the non-cyclic interpretation applies *HD* to another application of Y. In the latter case five more reduction steps are necessary (Y, U, B, K and Bp) to evaluate a new copy of the tuple cell, hence work is duplicated. This demonstrates that the cyclic interpretation of the Y rewrite brings an advantage to the handling of multiple *WHERE* clauses. With the non-cyclic interpretation work is done more than once

hence full laziness is lost.

4.3. Standard strict combinators

The standard strict combinators are shown in figure (26). Strict argument positions are flagged with an asterisk (*). Such arguments are evaluated till the head normal form is reached. A head normal form is either a scalar (e.g. a number), a pair (:) or a combinator that is applied to insufficient arguments. Nor these arguments nor the head and the tail of a pair are evaluated. Arguments or functions enclosed in sharp brackets (⟨ ⟩) have a special significance, which is expressed in an informal way, including the required types of the arguments. An exclamation mark (!) denotes a non-standard evaluation strategy that is explained separately in the text.

AND TRUE* a	→a
AND FALSE* a	→FALSE
APPEND NIL* b	→b
APPEND (a : x)* b	→a : (APPEND x b)
ARCTAN a*	→⟨arc tangent a⟩
CHAR ⟨character denotation⟩*	→TRUE
CHAR ⟨not character denotation⟩*	→FALSE
CODE a*	→⟨number to character conversion a⟩
COND TRUE*	→K
COND FALSE*	→(K I)
COS a*	→⟨cosine a⟩
DECODE a*	→⟨character to number conversion a⟩
ENTIER a*	→⟨entier a⟩
EXP a*	→$\langle e^{a} \rangle$
FDIV a* b*	→$\langle \frac{a}{b} \rangle$
FUNCTION ⟨function application⟩*	→TRUE
FUNCTION ⟨not function application⟩*	→FALSE
GR ⟨number > b⟩* b*	→TRUE
GR ⟨number ≤ b⟩* b*	→FALSE

GRE ⟨number ≥ b⟩* b* →TRUE
GRE ⟨number < b⟩* b* →FALSE

HD (a : b)* →a

INTDIV a* b* →⟨a DIV b⟩

LIST (a : x)* →TRUE
LIST NIL* →TRUE
LIST ⟨not (constructor or NIL)⟩* →FALSE

LOG a* →⟨natural logarithm a⟩

LOGICAL TRUE* →TRUE
LOGICAL FALSE* →TRUE
LOGICAL ⟨not (TRUE or FALSE)⟩* →FALSE

MINUS a* b* →⟨a - b⟩

MOD a* b* →⟨a MOD b⟩

MUCHGR a* ⟨number x such that a - x = a⟩* →TRUE
MUCHGR a* ⟨number x such that a - x ≠ a⟩* →FALSE

NEG a* →⟨-a⟩

NOT TRUE* →FALSE
NOT FALSE* →TRUE

NUMBER ⟨number denotation⟩* →TRUE
NUMBER ⟨not number denotation⟩* →FALSE

OR TRUE* b →TRUE
OR FALSE* b →b

PLUS a* b* →⟨a + b⟩

POWER a* b* →$\langle a^b \rangle$

READ ⟨at end of file⟩ →NIL
READ ⟨not at end of file⟩ →⟨character from file⟩ : READ ⟨file⟩

SIN a* →⟨sine a⟩

SQRT a* →$\langle \sqrt{a} \rangle$

TIMES a* b* →⟨a × b⟩

TL (a : b)* →b

WRITE a b	→(((%W:%R:%I:%T:%E:% :NIL):a):b)
(a : x)* ⟨1⟩*	→a
(a : x)* ⟨number b > 1⟩*	→x ⟨b - 1⟩

Figure 26 : Standard strict combinators

4.4. The equality test

The equality test serves to determine whether two combinations represent the same data structures. The arguments to the *EQ*-combinator (see figure 27) are normalised during the comparison, until either both are in normal form, or two corresponding components of the argument structures compare unequal. The only constructor that is allowed is the list constructor (:). The comparison of normal forms that are curried applications of combinators yields *FALSE*.

There is one exception to the above rule. The equality test yields *TRUE* if the structures compared refer to the same node in the graph. The subgraph rooted at such a node may even represent a reducible expression, which is not evaluated. For example the comparison in *if ones = ones then* ··· *WHERE ones* = 1 : *ones* yields *TRUE*, while it should have evaluated to ⊥. This is what happens when we reduce *if ones = ones' then* ··· *WHERE ones* = 1 : *ones*; *ones'* = 1:*ones'*. It should be noted that although the answer given to the former equality test is not incorrect, it is inconsistent with the latter test, hence it should be rejected.

Turner has probably introduced this exception as an optimisation, because it avoids comparison of two data objects when they are represented by one and the same graph. Even if the optimisation would include something like a check on the type of the root of the common graph, counter examples against the correctness of the optimisation can still be constructed. The only water proof method is to verify that the entire common subgraph is free of applications, which completely defeats the optimisation.

EQ a! a!	→TRUE
EQ a! ⟨structure ≠ a⟩!	→FALSE

Figure 27 : The equality test

4.5. Pattern matching combinators

Pattern matching on arguments of user defined functions is implemented by *TRY*, *MATCH* and the strict version *Us* of the uncurry combinator. Alternative clauses in a user defined function are combined with applications of *TRY*. The *MATCH*- and *Us*-

combinators perform the necessary evaluation to select the appropriate alternative. The abstraction and transformation rules for the combinators *MATCH*, *TRY* and *Us* are shown in figure (28). An argument pattern that contains constants or multiple occurrences of the same variable is first transformed to one that contains unique variables only, before the abstraction can be performed. The transformation adorns the function body with curried applications of *MATCH* to implement the required pattern matching. The arguments are scanned from right to left for constants and multiple occurrences of variables. A constant (c) is replaced by a new variable (say z) and the current defining expression (E) by (*MATCH* c E z). Each occurrence of a variable (x) that has already been encountered in an argument pattern is replaced by a new variable (y) and the current defining expression (E) by (*MATCH* x E y). The resulting expression can then be abstracted using Turner's rules.[4] Function arguments, which may consist of arbitrarily complex patterns, are abstracted out of the function body, starting with the right most argument. The transformation rule for *TRY* combines two defining clauses of a user function once the arguments have been abstracted out of each clause separately. The rule is applied repeatedly, starting with the first two clauses, until the expressions in all clauses with the same (function) name are combined into a single combinator expression. The abstraction rule for *Us* is similar to that of *U*.

f ... c ... = E	⇒ f ... z ... = MATCH c E z
f ... x ... x ... = E	⇒ f ... x ... y ... = MATCH x E y
f = E_1 f = E_2	⇒ f = TRY E_1 E_2
[x:y] E	⇒ Us ([x] ([y] E))

Figure 28 : Pattern matching transformation and abstraction rules

Figure (29) shows the rewrite rules for the pattern matching combinators. The comparison performed by the *MATCH*-combinator is the same as that performed by the equality test (*EQ*). *MATCH* compares its first and third arguments and normalises these as the comparison proceeds, but the remaining (second) argument is left for what it is. An unsuccessful comparison causes the application to yield the constant *FAIL*, which is introduced specifically to convey information from *MATCH* and *Us* to *TRY*. The *Us*-combinator is used to apply a function (f in figure 29) to the head (a) and tail (x) of a list structure. The difference between *U* and *Us* is that the latter yields *FAIL* if its first argument does not represent a list constructor. This notifies *TRY* of the fact that an argument (to a user defined function) does not have the appropriate structure.

The *U*-combinator itself does not require its argument to be a list constructor. Instead it applies *HD* and *TL* on its argument (see figure 20). Here *HD* or *TL* will signal a run time error if their argument is not a list.

MATCH a! f a!	→f
MATCH a! f ⟨structure ≠ a⟩!	→FAIL
TRY f g x	→TRY (f x) (g x)
TRY (FAIL ...)* g	→g
TRY ⟨data⟩* g	→⟨data⟩
TRY ⟨other⟩* g	→"can not evaluate TRY ⟨other⟩ g"
Us f (a : x)*	→f a x
Us f ⟨not a constructor⟩*	→FAIL

Figure 29 : Pattern matching combinators

The *TRY*-combinator is an exceptional one in the context of fixed arity combinators, because there really are two versions of it with the same name. The appropriate version is selected based on the available number of arguments. If the *TRY*-combinator is called with three arguments, as the first step it pushes references to the third argument into the graphs of the first two. The sharing of the third argument x is not clearly shown in figure (29), but the pictures of figure (31) show this in a better way. For the moment we will assume, that no more than three arguments are initially available to *TRY*, hence an application with two arguments is what remains. We will consider the extension to the case of more arguments shortly. The next step in the evaluation is to bring the first argument, in figure (29) this is (f x), to head normal form. Depending on the result of this evaluation there are three possible routes for the evaluation of *TRY* to proceed. If the second argument is:

FAIL or an application of *FAIL*:

> During normalisation of the first argument an application of *MATCH* or *Us* has signalled, that the expression to be matched did not fit the required pattern. This means that an alternative clause must be investigated. In a SASL function with more than one argument, failure to match any but the last argument produces an application of *FAIL* to the remaining arguments.

data; a constant or list constructor:

> The normalisation has produced a value, either a scalar or a list, that will serve as the result of the pattern match.

an application of something else than *FAIL*:

> If a user defined function is supplied with insufficient arguments, the curried

version of the function still requires applications of *TRY* to deal the missing arguments. These can never be supplied any more by the reduction process, hence an error message is printed. However a combinator reducer is normally embedded in a conversational system, that allows the user to repeatedly present function applications for evaluation. In this context it is possible to launch the evaluator a second time on the same function but with the missing arguments supplied. In our experiments this is not the case.

The entire sequence of actions as described above is counted as one reduction step.

Figure (30) shows an example of a SASL function *F* with three alternative clauses and the combinator code that appears during the evaluation of (*F* 1) and (*F* 3). The former yields a successful pattern match whereas the latter demonstrates the use of *FAIL*. The sharing of the arguments normally created by *TRY* is not shown in the example.

defining clause 1	*F* 1 = 10
defining clause 2	*F* 2 = 20
defining clause 3	*F a* = *a*
compiled code	(TRY (TRY (MATCH 1 10) (MATCH 2 20)) I)
application	*F* 1
step 1	(TRY (TRY (MATCH 1 10) (MATCH 2 20)) I) 1
step 2	(TRY (TRY (MATCH 1 10) (MATCH 2 20) 1) (I 1))
step 3	(TRY (TRY (MATCH 1 10 1) (MATCH 2 20 1)) (I 1))
result 3a	(TRY (TRY 10 (MATCH 2 20 1)) (I 1))
result 3b	(TRY 10 (I 1))
result 3c	10
application	*F* 3
step 1	(TRY (TRY (MATCH 1 10) (MATCH 2 20)) I) 3
step 2	(TRY (TRY (MATCH 1 10) (MATCH 2 20) 3) (I 3))
step 3	(TRY (TRY (MATCH 1 10 3) (MATCH 2 20 3)) (I 3))
result 3	(TRY (TRY FAIL (MATCH 2 20 3)) (I 3))
step 4	(TRY (MATCH 2 20 3) (I 3))
result 4	(TRY FAIL (I 3))
step 5	(I 3)
result 5	3

Figure 30 : Pattern matching with *TRY* and *MATCH*

The extension referred to earlier of pattern matching to (user defined) functions with more than one argument does not require more than a slight change in the way *TRY*'s first step operates. Currently the *TRY* rule with two arguments is applied immediately after the first rewrite with three arguments. Instead we will perform the rewrite with three arguments repeatedly until no more arguments are available. Consider as an example the case of figure (31), where *TRY* is presented with four arguments, two of which (f and g) represent alternatives in the body of a user defined function, whereas x and y refer to the arguments of the user defined function.

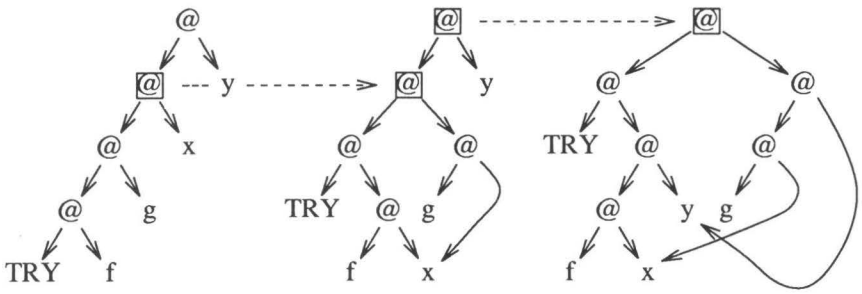


Figure 31 : The *TRY*-combinator with four arguments

During the evaluation two nodes are overwritten. This is necessary to guarantee full laziness, as these nodes (indicated by boxes in figure 31) may be shared from other parts of the graph. The top node of the entire subgraph is even overwritten a second time, during the final rewrite step (not shown in the figure), when (f x) has been normalised.

The *TRY*-combinator causes some inconvenience in a frame work of fixed arity combinators. The first problem is the way reduction steps are counted. Most of the statistics in our work are based on counting reduction steps. For all combinators except *TRY*, there is a bounded amount of work associated with each reduction rule, if we disregard the work performed by the storage allocator for the moment. Furthermore, the rewrite rules associated with the other combinators do not differ much, such that we may assume that each requires about the same amount of work. Since *TRY* accepts any number of arguments, there is no a priori upper bound on the amount of work involved in performing the rewrite. However, few programmers would use functions with more than say a dozen arguments. Hence there is a practical upper bound on the amount of work involved. We have observed that for a bench mark of SASL programs,[11] the number of arguments to *TRY* does not exceed 6 with an average of 3.3 arguments per application. We have also found that less than 10% of all combinators executed are *TRY*-combinators, hence the problem is not too serious. However, it is

something to be continually aware of when performing experiments with real programs.

A second problem is, that occasionally all nodes in the graph that are still in use must be visited. This happens for example during the scan or copy phase of a non-reference count garbage collector. However, while a reduction step is in progress, the graph may be temporarily disconnected. A *TRY*-reduction for example first claims a new node to combine the first and third arguments to a new application (the application $(f\ x)$ in figure 31). If by then the storage allocator has run out of available nodes, a sweep of the garbage collector is necessary. Since the most recently claimed node is not yet reachable from anywhere else in the graph, we must remember somehow not to recuperate that node as garbage.

A convenient solution to this problem is to perform garbage collection and similar activities only before starting a new reduction step, when the graph is connected. With fixed arity combinators there is a guarantee that sufficient free nodes are available while the reduction step is in progress. This method can still be applied with the *TRY*-combinator present, but the threshold on the number of nodes that must be available before a reduction step may be started, is application dependent. Another solution is to check before each iteration of *TRY*'s first step if three more nodes are still available.

4.6. Summary

In the literature some aspects of Turner's method for combinator graph reduction are not described. We have shown that the cyclic interpretation of the Y-reduction is necessary to implement *WHERE* clauses with multiple definitions efficiently. With a non-cyclic interpretation full laziness is lost. The second aspect concerns pattern matching on arguments to user defined functions. Pattern matching is supported by three combinators: *TRY*, *MATCH* and *Us*. Two alternatives to a function are combined by applications of *TRY* to a new alternative. *MATCH* performs the equality test necessary to determine whether a given argument to a function matches the pattern of a particular argument and clause. The *Us*-combinator is used to support pattern matching on arguments that are lists. The *TRY* combinator does not fit the framework of fixed arity combinators, because it can be supplied with an arbitrary number of arguments. This makes it more difficult to control allocation of storage.

Chapter 5

Implementation aspects of Turner's method

A reduction model based on Turner's method of combinator graph reduction leaves a large degree of freedom to the implementation that can be exploited to make garbage collection faster. In this chapter we will discuss the implementation aspects that influence the rate at which storage cells are claimed and the difficulty with which cells can be recuperated. We show how the claim rate of cells can be lowered by increasing the amount of sharing. In the previous chapter we saw that the presence of cycles in combinator graphs forms an obstacle to efficient garbage collection. In the current chapter we present a method that allows graph traversal algorithms to detect cheaply whether a cycle is being traced. Although effective in most cases the method is not applicable to all kinds of cycles.

We first briefly review the main task of the reducer. A reducible expression is represented by a "spine" of binary application nodes that are strung together via their left pointer fields. The right pointer of a node refers to the argument of the function that is represented by the left descendant. The "unwind" algorithm discovers the next (normal order) redex. It may be started with a reference to the root of any (sub)graph. Initially the unwind algorithm is started with the root of the main application. The algorithm chases the left pointers of the application nodes until a combinator or constructor node (*pair*) is found. The combinator or list selection is fired if sufficient arguments are available. Otherwise a partial application has been discovered. Excess application nodes are supposed to be handled by combinators that will be fired later. Strict combinators require the use of the unwind algorithm to normalise their strict arguments. Recursive invocations of the unwind algorithm are therefore interspersed with suspended firings of strict combinators.

5.1. Avoiding the disadvantages of cycles in combinator graphs

In pure graph reduction, recursion is implemented by using cyclic graphs. The presence of cycles complicates the algorithms necessary to manipulate the graphs that occur during reduction. The main issue is how to detect cycles during traversal of the graph. Problems exist in the areas of garbage collection[20] the distribution of parallel computations through copying of graphs,[14, 15] and other areas such as the detection of cycles when unwinding the spine or printing subgraphs in error messages. One way to avoid cycles is not to use graphs. Wadler[27] describes a class of functional programs that can be executed without ever requiring the use of the graph. A non-restrictive

approach is to avoid the use of the graph whenever possible. This method is used by most contemporary implementations of sequential reduction.[5,6,8] In our own proposal for a parallel graph reduction system a stand point between the two is taken. When evaluating sequentially we use Turner's method, hence graphs are not avoided. For parallel evaluation we use trees rather than graphs. Application programs that can not be written in the appropriate form can not be evaluated efficiently in parallel, but they can still be evaluated sequentially. The restrictions that we impose on application programs have no consequences other than efficiency limitations. In Vree's work[28,29] the concept of the "essential" cycle is introduced and transformation methods are developed to avoid essential cycles in functional programs. A cycle such as that occurring in the Hamming problem[30] is an essential cycle. The cycles that we have encountered in the previous chapters are inessential. In the remainder of this section we are concerned with avoiding a disadvantage of inessential cycles in Turner's combinator graph reduction system.

In Turner's implementation there are two sources of cycles. One is the linkage of the main combination and the user defined functions (see chapter 4) and the other is the cyclic rewrite rule for the Y-combinator. The former cycle will be called a "linkage cycle", the latter a "Y-cycle". Both originate from the use of recursive functions in the SASL programs (except non-recursive multiple *WHERE* definitions as explained earlier). The distinction is a bit artificial, because the linkage cycles could be replaced entirely by Y-cycles, if the program were given the form of a main expression with all global functions defined under a single *WHERE*. In that form, a large program would become grossly inefficient, because of the massive amount of tupling and untupling that is required to access the individual functions. We will show that the linkage cycles can be used to our advantage.

Y-cycles are created and destroyed during reduction. Linkage cycles are more tractable, because there is a unique name (of a globally defined function) associated with each linkage cycle and the name is permanent. Permanent means, that the situation can not be changed by the reduction process because it is referentially transparent. The way to let a graph traversal algorithm know whether a cycle is being traced is by marking the root of the subgraph that represents a global function definition. There exists a 1-1 correspondence between such roots and global function names. These marks will be called "cut marks" to reflect the intention that a cycle may be cut at this point. The marking can be performed almost free of charge during the linkage of the global functions and the main expression. The "cycle closing pointers" for linkage cycles are always incident upon a node with a cut mark. Although the intention is the same, the cut mark mechanism is different from for instance Brownbridge's method of using weak and strong pointers[20] because cut marks are permanent. A minor difference is the fact that we mark the node rather than the pointer.

The cut mark mechanism can not be used with the Y-cycles, but it is possible to transform local function definitions into global function definitions by means of lambda lifting.[31] This transformation incurs a certain cost if free variables occur in a *WHERE* expression. Lambda lifting binds free variables to new arguments, such that the newly created global functions will have more arguments than their local counter parts. The invocations of the former *WHERE* functions must be adorned with more actual arguments as well.

We will give an example of the transformation by lambda lifting and quantify the cost involved. Figure (32-a) presents a function to search a *list* for the occurrence of a particular *item*. The argument *item* of the global function *search* occurs as a free variable in the body of the local function *try*. Figure (32-b) shows the lambda lifted versions *search'* and *try'*.

```
search item list          =  try list 1
                             WHERE
                             try () i                =  0
                             try (head : tail) i     =  item = head → i
                                                        try tail (i + 1)
```

(a) *search* with a recursive *WHERE* expression

```
search' item list         =  try' list 1 item
try' () i item            =  0
try' (head : tail) i item =  item = head → i
                             try' tail (i + 1) item
```

(b) The lambda lifted version of *search*

Figure 32 : Lambda lifting

The untransformed function *search* prepares a specialised copy of *try* when invoked to search for a certain *item*. This version of *try*, with that certain *item* built in, is reused during each recursive invocation on the next element of the *list*. The transformed version does not have this opportunity. Each recursive invocation of *try'* requires the *item* as an argument. Compiled into Turner's combinators the untransformed version requires 13 reduction steps per recursive invocation; the transformed version 16. At the cost of some extra reduction steps we are thus able to transform a program with untractable Y-cycles into an equivalent one with permanent tractable cycles. The cost of lambda lifting on our benchmark of functional programs

was found to increase the number of reduction steps by less than 10%, while the number of cycles was reduced dramatically.[11]

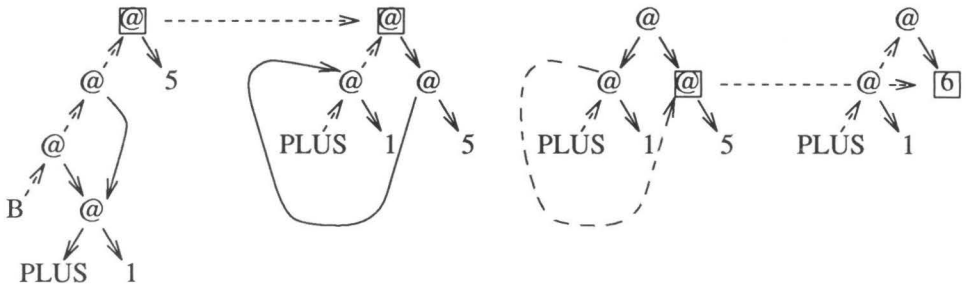
The lambda lifting transformation allows complete freedom in the ordering of arguments. For example in figure (32-b) we could have inserted *item* as the first argument in stead of the last to yield *try''*. This influences the combinator code in a negative way. Each recursive invocation of *try'* requires 13 reductions steps versus 15 for *try''*. A similar effect can be noticed during normal program development. The performance of SASL programs is sensitive to the ordering of arguments.

5.2. The left ancestors stack and pointer reversal

The selection of the next reducible expression with normal order evaluation is conveniently controlled with a left ancestors stack. It holds the pointers to the spine of application nodes that define the current redex (and the redexes that are suspended). The stack itself can be implemented by means of pointer reversal[32] or by using a contiguous area of store that is large enough to hold the biggest stack that may occur. The former method has the advantage, that no a priori upper bound on the size of the stack has to be known. It has also disadvantages. For instance the arguments of the current combinator can not be accessed at constant cost as with the real stack. The reversed pointers need to be chased, starting with the first argument until the required one is found. A more serious disadvantage, which is often overlooked, is that the chain of reversed pointers must be rebuilt whereas the appropriate cells in the real stack can be updated as reduction proceeds. Furthermore, the unwind of the real stack need only be performed once per redex, but with the reversed pointer stack the (rewind/)unwind must be performed once more for each strict argument. The reason is, that a curried application of a strict combinator may be shared by one of its arguments. The example of figure (33) may serve to illustrate this point.

```
inc x      =  1+x
double y   =  inc (inc y)
double 5
```

(a) Function definitions and main application



(b) Reduction steps and pointer reversal

Figure 33 : Pointer reversal for curried strict combinators

After the first (*B*) reduction step, the chain of reversed pointers necessary for the left most application of *PLUS* links the left descendant of the root to the root itself (second graph in figure 33-b). Since *PLUS* also requires a value for its second argument, it causes the second application of *PLUS* to be fired after the original pointers have been re-reversed. Unwind re-re-reverses the pointers to provide access to the arguments 1 and 5. Then it creates the situation shown in the third graph. The last graph shows the state of the pointers after they have been re-re-re-reversed for the suspended application of *PLUS* to be resumed. Stoye,[33] without mentioning it, solves this problem at the cost of an extra "pcomb" node that represents the curried application of *PLUS*. This node also serves the purpose of saving the state of the suspended computation, which Turner's implementation stores on the regular recursion stack of the reduction algorithm (unwind).

5.3. Higher order sharing

In the graphical representation of combinations the application and constructor nodes form the interior nodes. The leaf nodes represent constants of various types: combinators, characters, booleans, *NIL*, *FAIL* and the (floating point) numbers. Except the numbers there are few elements in each of these sets. This opens the possibility to optimise the use of storage for the representation of constants. Two ways of storing constants can be envisaged. The direct access method stores a constant in the space normally reserved for a pointer. Tag bits are used to distinguish between a pointer and a data item. The indirect access method creates a separate node for a constant. This time the node is tagged to make the distinction between an interior and a leaf node. Since a pointer generally requires fewer bits to store than a floating point number, the direct access method only works for floating point numbers if nodes of different sizes are allowed. This makes the storage allocator more complicated and increases the cost

of garbage collection. Although the other constants do not pose a problem of the sort (strings are stored as lists of characters), we will not consider the direct access method further.

The indirect access method appears wasteful at first, but there is a simple optimisation possible that makes it more attractive. For the non-numeric constants we keep an array of pointers to nodes with the appropriate values. Instead of creating a new node, the pointer to the only copy of the required node is used. In a referentially transparent system no harm can be done by this form of sharing that we will call hyper sharing. The method was used by Turner in his implementation of SASL. The idea can be extended to numbers by using a hash table, or as we have done, by storing the pointers to certain numbers (the integers from -128 to 127) in an array. For our benchmark of functional programs we found that depending on the application program, the indirect access method saves 20% to 45% of the space. These savings are largely due to hyper sharing of non-numeric nodes, in particular the combinators. Savings due to hyper sharing of numbers turns out to be insignificant except for a numeric application (tidal simulation), where it amounts to 17%. We do not expect a difference in execution speed between systems with and without hypersharing. In either case dereferencing pointers to a node will incur the same cost. Creation of a new node without hyper sharing requires a similar amount of work as for instance a table lookup necessary for hyper sharing.

In a referentially transparent system, there are other ways of saving space by higher order sharing. For instance the equality test in SKIM[32] creates "super" sharing by commoning up equal structures. We have investigated another possibility of super sharing. For instance the combinator *SQRT* if applied to 0 will yield 0. With super sharing, the rewrite will yield a reference to the argument rather than create a new 0-node. The root of the *SQRT* application is changed into an indirection to its argument rather than into a node that represents the zero value itself. Other combinators that allow for super sharing are *ENTIER*, *FDIV*, *INTDIV*, *MINUS*, *MOD*, *NEG*, *PLUS*, *POWER* and *TIMES*. From experiments with our benchmark of SASL programs we found the savings in space to be less than 5%. The implications on processing time have not been investigated, but they are not expected to be significant either. The savings that can be achieved with combinators operating on more complex data structures are more important. This is why we have experimented with combinators for the support of arrays. These are described in section 5.5.

5.4. Avoiding the production of garbage

Reference counting is an attractive garbage collection method because it allows the mutator to detect whether nodes that are of current interest are shared or not. Such

knowledge opens the possibility for immediate recycling of nodes. For instance the S rewrite rule requires two new nodes, while there are two nodes on the spine that can be recycled immediately if they are not shared. The array combinators such as *change* and *exchange* allow for even greater savings, since a complete array structure may be destructively updated if it is not shared. We are currently investigating the savings that may be obtained by immediate recycling of nodes.

With reference counting it is easy to detect whether a node is shared, but there is no inexpensive way to find out from which nodes the pointers emanate. With this information complete avoidance of indirection nodes would be possible. From experiments with our benchmark of SASL programs we know, that the root of the subgraph that represents the current redex is often shared. Hence the root indirection does not disappear immediately following the next rewind. In our benchmark there is one application that generates slightly more than 10% indirection nodes, the others produce about half that percentage. Hence avoiding indirection nodes can not yield large savings in either space or time, but large enough to warrant further investigation. From our experiments we also found that less than 10% of all nodes are shared, with an average reference count of about 1.2. This inspired us to experiment with a new implementation of reference counting that maintains an "edge list" as follows: to each shared node x attach a list of pointers to all the nodes that have a pointer to x. The number of elements in the list is equal to the ordinary reference count. With this list it is easy to find the direct ancestors to a node. The edge list will increase the total space requirements by at least 20%, since there are 1.2 times as many pointers as there are nodes. The increase in processing time is likely to be even more than 20%; we found that on the average 3 times as many edges are created than nodes. By coincidence the S-rewrite has an average behaviour in this respect: it claims two new nodes and needs 6 new edges. We may conclude that an edge list, implemented as proposed above, is not likely to either make a reducer faster or lower its space requirements.

5.5. Array combinators

To gain experience with the use of arrays in a functional context, the concept of a linear array has been incorporated in the reduction model, supported by a number of array operations. In the proposal by Barendregt & van Leeuwen[34] arrays can be of arbitrary dimension. In order not to complicate the implementation of a new concept before experience has been gained with it, we decided to support linear arrays only. A component of an array may be any object, including an array. For the same reason we have not implemented the arrays with constant access times to the elements. The standard list constructor is used as the basis for array data types. The implementation of arrays is an example of hybrid simulation. Rather than providing a truly efficient implementation of some component we replace it by a prototype and abstract away

from its deficiencies by relating the performance of the component to (abstract) reduction steps rather than (concrete) seconds.

The head of the list that represents an array forms its descriptor. The descriptor consists of a pair of integers: the lower bound and the upper bound of the array. The tail of the list forms a chain of pair nodes that keeps the actual array elements together. This chain is called the spine of the array. A full complement of array combinators is proposed and arrays should only be accessed via one of the 13 array combinators as shown in figure (34). In the experimental implementation array access via list selection, *HD*, *TL* etc. is possible, but not advisable, since an efficient array implementation will probably not be based on the standard list constructor. There are four categories of array combinators:

Creation

The combinators *cumulate*, *makerow* and *tabulate* are used to create an array from arbitrary functions and data.

Element Access

The combinators that must be used to access array elements are *subscript*, *change* and *exchange*. The latter two are included to allow for experiments with destructive updating of arrays, which is perfectly acceptable if the array is not shared.

Transformation

Arrays can be manipulated as a whole by the combinators *concatenate*, *for*, *split* and *reverse*.

Descriptor access

The descriptor of an array can be accessed via the combinators *lwb*, *upb* and *descriptor*.

The implementation of arrays guarantees that the length of the spine of an array is equal to the number of elements defined by the descriptor (*upperbound* – *lowerbound* + 1). Hence the *NIL* at the end of the spine is redundant. As a consequence, there are infinitely many "null" arrays that satisfy the invariant. The array combinators accept null arrays as arguments. There is for instance no harm done in concatenating a null array onto another array or reversing a null array. It is considered an error to select non-existent elements from an array.

change ((lb : ub) : a_{lb} .. a_n .. a_{ub} : NIL)! n* x
→((lb : ub) : a_{lb} .. x .. a_{ub} : NIL)

concatenate ((la : ua) : aa)! ((lb : ub) : ab)!
→((la : ⟨ua+ub+1-lb⟩) : ⟨aa++ab⟩)

cumulate f NIL*
→((1 : 0) : NIL)

cumulate f ⟨structure x ≠ NIL⟩*
→((1 : ⟨n+1⟩) : x : (f x) :
.. (f^n x) : NIL)

descriptor (d : a)!
→d

exchange ((lb : ub) : a_{lb} .. a_m .. a_n .. a_{ub} : NIL)! m* n*
→((lb : ub) : a_{lb} .. a_n .. a_m .. a_{ub} : NIL)

for ((lb : ub) : a_{lb} .. a_{ub} : NIL)! f
→((lb : ub) : (f a_{lb}) .. (f a_{ub}) : NIL)

lwb ((lb : ub) : a)!
→lb

makerow (a_1 : a_2 .. a_n : NIL)! lb*
→((lb : ⟨lb+n-1⟩) : a_1 : a_2 .. a_n : NIL)

reverse ((lb : ub) : a_{lb} .. a_{ub} : NIL)!
→((lb : ub) : a_{ub} .. a_{lb} : NIL)

split ((lb : ub) : a_{lb} .. a_n : a_m .. a_{ub} : NIL)! n*
→((lb : n) : a_{lb} .. a_n : NIL) :
((lb : ⟨lb+ub-n-1⟩) : a_m .. a_{ub} : NIL)

subscript ((lb : ub) : a_{lb} .. a_n .. a_{ub} : NIL)! n*
→a_n

tabulate f (lb : ⟨number ub < lb⟩)**
→((lb : ⟨lb-1⟩) : NIL)

tabulate f (lb : ⟨number ub ≥ lb⟩)**
→((lb : ub) : (f lb) : (f (lb+1))
.. (f ub) : NIL)

upb ((lb : ub) : a)!
→ub

Figure 34 : Linear array combinators

The reduction strategy for arrays is such, that the descriptor and the spine of an argument flagged with an exclamation mark are normalised. The actual array elements are evaluated under the normal lazy regime. The double asterisk (**) signifies that the annotated descriptor is normalised. An index operation on an array requires one reduction step in addition to those necessary to normalise the index value. The real execution times depend on the number of list elements that have to be skipped, but the reduction step statistics are representative for a proper implementation of linear arrays. The *cumulate* combinator reduces x, $(f\ x)$, $(f\ (f\ x))$ etc. until the result of such an application is $(f^{n+1}\ x) \equiv NIL$. The length of the array is given by the number of non-*NIL* applications.

The experience that we have gained with the array combinators is restricted to two application programs: a simulation of the tidal waves in the North Sea and the fast Fourier transform.[14] The applications have been run in two modes: with arrays implemented as described above and with arrays implemented as ordinary SASL functions. Both implementations use lists to store the components of the array. The number of reduction steps required gives an indication of the performance improvement that an optimised array implementation may be expected to offer. However, with powerful operations such as *split* accounted for as a single reduction step, this can no longer be considered a uniform performance measure. From the difference in reduction steps we may calculate an upperbound on the performance gain that may result for this particular program if arrays are implemented efficiently. The gain that we have measured amounts to 57% fewer reduction steps for the tidal simulation and 87% fewer reduction steps for the Fourier transform. The *subscript* combinator represents 97% respectively 83% of all the array combinators executed. With an optimised implementation and a more uniform performance measure we may still hope for good results.

5.6. Conclusions

Most cycles that occur during combinator graph reduction are inessential. Depending on the source of cycles we distinguish between tractable and untractable cycles. Tractable means that one of the nodes involved in a cycle can be permanently marked as such. Graph traversal algorithms can be made to benefit from this information. Suitable modification of the abstraction algorithm, the rewrite rule for the Y-combinator, and graph traversal algorithms embedded in the reducer avoids untractable cycles. Of the inessential cycles those that are produced by the Y-reduction can be replaced by cycles that are tractable at the cost of a small increase in reduction steps. Johnsson's lambda lifting technique is necessary to avoid untractable cycles.

Methods to reduce the space requirements of fixed combinator graph reduction include successful and less successful ones:

- Hyper sharing of scalars may reduce the space requirements of functional programs. Super sharing of scalars does not bring a significant advantage.
- Maintaining an edge list does not improve the speed of a reducer nor reduce its space requirements.
- Array combinators provide useful support for certain types of algorithms and will give rise to faster sequential reduction with such algorithms.

References

1. J. Cohen, "Garbage collection of linked structures," *Computing Surveys* **13**(3) pp. 341-367 (Sep. 1981).
2. D. Ungar, "Generation scavenging: a non-disruptive high performance storage reclamation algorithm," *Software engineering symp. on practical software development environments, SIGPLAN Notices* **19**(5) pp. 157-167 ACM, (Apr. 1984).
3. D. A. Turner, "SASL language manual," Technical report, Computing Laboratory, Univ. of Kent at Canterbury (Aug. 1979).
4. D. A. Turner, "A new implementation technique for applicative languages," *Software Practice and Experience* **9**(1) pp. 31-49 (Jan. 1979).
5. 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 *Third conf. on functional programming languages and computer architecture, LNCS 274*, ed. G. Kahn,Springer verlag, Portland, Oregon (Sep. 1987).
6. J. Fairbairn and S. Wray, "Tim: A simple lazy abstract machine to execute supercombinators," pp. 34-45 in *Third conf. on functional programming languages and computer architecture, LNCS 274*, ed. G. Kahn,Springer verlag, Portland, Oregon (Sep. 1987).
7. R. J. M. Hughes, "Super combinators - A new implementation method for applicative languages," pp. 1-10 in *ACM symp. on Lisp and functional programming*, ACM, Pittsburg, Pennsylvania (Aug. 1982).
8. T. Johnsson, "Efficient compilation of lazy evaluation," *Sigplan Notices* **19**(6) pp. 58-69 (Jun. 1984).
9. A. J. Field and P. G. Harrison, *Functional programming*, Addison Wesley, Reading, Massachusetts (1988).
10. S. L. Peyton Jones, *The implementation of functional programming languages*, Prentice Hall, Englewood Cliffs, New Jersey (1987).
11. P. H. Hartel and A. H. Veen, "Statistics on graph reduction of SASL programs," *Software practice and experience* **18**(3) pp. 239-253 (Mar. 1988).
12. P. H. Hartel, "A comparative study of three garbage collection algorithms," PRM project internal report D-23, Dept. of Comp. Sys, Univ. of Amsterdam (Feb. 1988).
13. P. H. Hartel, "The average size of ordered binary subgraphs," in *Graph-theoretic concepts in computer science, Int. workshop WG'88, to be published in LNCS*, ed. J. van Leeuwen,Springer Verlag, Amsterdam, Netherlands (Jun. 1988).

14. W. G. Vree and P. H. Hartel, "Parallel graph reduction for divide-and-conquer applications Part I - program transformation," PRM project internal report D-15, Dept. of Comp. Sys, Univ. of Amsterdam (Apr. 1988).

15. P. H. Hartel and W. G. Vree, "Parallel graph reduction for divide-and-conquer applications Part II - program performance," PRM project internal report D-20, Dept. of Comp. Sys, Univ. of Amsterdam (Apr. 1988).

16. H. Glaser, C. Hankin, and D. Till, *Principles of functional programming*, Prentice Hall, Englewood Cliffs, New Jersey (1984).

17. H. P. Barendregt, *The lambda calculus, its syntax and semantics*, North Holland, Amsterdam (1984).

18. P. J. Landin, "The mechanical evaluation of expressions," *Computer Journal* **6**(4) pp. 308-320 (Jan. 1964).

19. H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeijer, and M. R. Sleep, "Term graph rewriting," pp. 141-158 in *Parle conf. on parallel architectires and languages, LNCS 259*, ed. J. W. de Bakker, A. J. Nijman, P. C. Treleaven,, Eindhoven, The Netherlands (Jun. 1987).

20. D. R. Brownbridge, "Cyclic reference counting for combinator machines," pp. 273-288 in *Second conf. on functional programming languages and computer architecture, LNCS 201*, ed. J.-P. Jouannaud,Springer verlag, Nancy, France (Sep. 1985).

21. D. P. Friedman and D. S. Wise, "Reference counting can manage the circular environments of mutual recursion," *Information processing letters* **8**(1) pp. 41-45 (Jan. 1979).

22. M. Schönfinkel, "Über die bausteine der mathematischen logik," *Mathematische Annalen* **92**(6) pp. 305-316 (1924).

23. Motorola Inc., *MC68000 8-/16-/32-Bit microprocessors programmer's reference manual*, Prentice hall, Englewood Cliffs, New Jersey (1986). fifth edition

24. N. Wirth, *Programming in Modula-2*, Springer verlag, Berlin (1982). second edition

25. R. F. H. Hofman, "An on-the-fly scheduling algorithm for an experimental parallel reduction machine," PRM project internal report D-18, Dept. of Comp. Sys, Univ. of Amsterdam (May. 1988).

26. D. A. Turner, "Another algorithm for bracket abstraction," *Journal of symbolic logic* **44**(2) pp. 267-270 (Jun. 1979).

27. P. Wadler, "Listlessness is better than laziness," pp. 45-52 in *ACM symp. on Lisp and functional programming*, ACM, Austin, Texas (Aug. 1984).

28. W. G. Vree, ''A parallel hydraulical simulation program in SASL,'' PRM project internal report D-13, Dept. of Comp. Sys, Univ. of Amsterdam (Oct. 1987).

29. W. G. Vree, ''Parallel graph reduction for communicating sequential processes,'' PRM project internal report D-26, Dept. of Comp. Sys, Univ. of Amsterdam (Aug. 1988).

30. D. A. Turner, ''The semantic elegance of applicative languages,'' pp. 85-92 in *Conf. on Functional programming languages and computer architecture*, ed. Arvind,ACM, Portsmouth, New Hampshire (Oct. 1981).

31. T. Johnsson, ''Lambda lifting: transforming programs to recursive equations,'' pp. 190-203 in *Second conf. on functional programming languages and computer architecture, LNCS 201*, ed. J.-P. Jouannaud,Springer verlag, Nancy, France (Sep. 1985).

32. W. R. Stoye, T. J. W. Clarke, and A. C. Norman, ''Some practical methods for rapid combinator reduction,'' pp. 159-166 in *ACM symp. on Lisp and functional programming*, ACM, Austin, Texas (Aug. 1984).

33. W. R. Stoye, *The implementation of functional programming languages using custom hardware,* Univ. of Cambridge, U.K. (Dec. 1985). PhD. Thesis

34. H. P. Barendregt and M. van Leeuwen, ''Functional programming and the language Tale,'' pp. 122-207 in *Current trends in concurrency: overviews and tutorials, LNCS 224*, ed. J. W. de Bakker, W.-P. de Roever, G. Rozenberg,Springer verlag, Noordwijkerhout, Netherlands (Jun. 1985).

FIVE PAPERS

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 18(3), 239–253 (MARCH 1988)

Statistics on Graph Reduction of SASL Programs

PIETER H. HARTEL
Computing Science Department, University of Amsterdam, Nieuwe Achtergracht 166, 1018 WV Amsterdam, Netherlands

AND

ARTHUR H. VEEN*
Computing Science Department, University of Nijmegen, Toernooiveld 1, 6525 ED Nijmegen, Netherlands

SUMMARY

The execution has been studied of four small and four medium-sized SASL programs, when interpreted by a variant of Turner's combinator reducer. Size, structure and composition of the combinator graph have been analysed at frequent intervals during the reduction process. The most interesting results are summarized and discussed. Nodes of the graph live rather short lives and are usually not shared. Cycles are rare, and linear lists are often short. In most aspects the behaviour of the graph is quite ordinary in the sense that a simple model is sufficient to obtain a good approximation.

KEY WORDS Combinators Graph reduction Statistical model

INTRODUCTION

In our project to design a parallel reduction machine for the efficient execution of functional programs,[1] we have become increasingly aware that the architecture should not just be language-based, i.e. exploit properties of the language for which it is designed, but also application based, i.e. exploit properties of the class of applications for which it is likely to be used.[2] However, the experience with functional languages is so limited that the nature, let alone the magnitude, of such properties is unclear. In the absence of definite data the architect often depends on intuition, but the intuition that we as programmers or as language implementers have developed in a mostly imperative world is a dangerous guide when it concerns the execution of functional languages implemented with lazy evaluation.

Already in our preliminary design effort we encountered the need for statistics about program properties. Most implementers of functional languages would agree that graph

* Current affiliation: Computing Science Department, University of Amsterdam.

0038–0644/88/030239–15$07.50

Received 12 March 1987
Revised 4 August and 15 October 1987

reduction, which evaluates shared subexpressions only once, is much more efficient than the conceptually simpler string reduction. One of the architectural options we were considering at the time was to forego locally the savings of graph reduction to reduce communication costs. In our analysis of this trade-off we needed to know the frequency and the distribution of shared subexpressions in a typical application program. We were surprised to find that the literature did not provide any such statistic.

Delimiting a sensible class of applications for which to design our machine is a difficult issue which we have not yet resolved to our satisfaction. In this paper we sidestep the issue by choosing a few SASL programs that we assume to belong to that class. Statistics will be presented that have been gathered from these applications by an interpreter especially instrumented to monitor the reduction process and to analyse the graph at regular intervals. Below a motivation is given for the choices that have been made concerning applications, programming language and method of translation. The methodology and the statistical results of the experiment are presented in subsequent sections.

Scope of the study

In the literature many functional languages have been reported, but very few sizeable applications. Most of these are written in SASL and since we have access to several of these programs as well as an implementation that could be easily instrumented to collect statistics, we have limited our study to SASL programs. We do not expect that the statistics reported here would be significantly different for other functional languages with lazy semantics, if implemented in a similar fashion.

The standard implementation of SASL is the combinator implementation described by Turner.[3] This implementation is the most suitable one for our purpose, since it contains very few optimizations that could bias our statistics. We used a locally produced variation, which employs the same set of combinators and abstraction rules and behaves identically to Turner's implementation with respect to the statistics reported in this paper. We repeated all measurements with another implementation that reduces the number of I-reductions by using a somewhat different allocation strategy and adds special combinators as debugging aids. This experiment was reassuring in the sense that all differences in the statistics could easily be related to the known differences in the implementations and none of the conclusions drawn in this paper were invalidated.

It is well known that the fixed set of simple combinators Turner uses leads to considerable overhead. Spectacular improvements have been reported either by using program-derived combinators or by avoiding graph transformations for those parts of the computation that can be performed eagerly.[4–6] We certainly plan to incorporate some of these improvements, or variations thereof, in our design, but which ones is not yet clear: the choice depends in part on the statistics reported here. For such improved combinator reducers many of our statistics would be quite different. However, since all combinator-based reducers reported so far can be considered to be optimizations of Turner's implementation, our statistics should still be useful as a basis for extrapolation or as a yardstick to measure progress.

The choice of the application programs has been *ad hoc*. We are only interested in an application program as far as it can be considered to be representative of a large class of applications. In our choice we are trying to approximate the average and avoid the unusual. We have a few dozen SASL programs at our disposal, most of them

small. Toy programs can usually be made to consume a considerable amount of computing time by providing them with appropriate input data, but we feel that this kind of extrapolation is a dangerous practice. Realistic programs not only represent a sizeable computation, but are also large in terms of program text. We are therefore mainly interested in larger programs. A few toy programs were included in our test set in an attempt to quantify our objection. Because the SASL interpreter that has been instrumented is not a particularly efficient one, collecting the statistics with a reasonable accuracy is a time-consuming process. Therefore only executions of at most a few hundred thousand reductions could be handled. This is equivalent to less than a minute CPU time on some of the most efficient implementations. We had access to four programs of about the right size. Three of them had the extra advantage that they have been described in the literature.

Related literature

In the past the study of compilation schemes and the composition and complexity of combinatory code has been given some attention. Almost always, worst-case complexity analysis is used, but some studies[7, 8] analyse the average case defined in a theoretical way by averaging over all possible expressions and assuming that all expressions are equally likely to occur. We have studied the average case in a practical sense, i.e. based on statistics gathered from programs occurring in practice.

The literature contains several sources of statistical data on programs, but none of the figures is directly applicable to graph reduction. Many data have been collected for imperative languages implemented on a conventional machine; Weicker[9] compiled a list of 16 collections on languages such as Fortran, PL/I, Algol68, Pascal and Ada. Because functional languages supposedly lead to a radically different programming style, their figures are not useful to us, with the possible exception of the relative frequency of arithmetic and comparative operations.

Studies of LISP programs may be slightly more useful, because LISP, although it contains imperative constructs, also has function application as its main construction device and binary lists as its main data structure. Gabriel[10] uses various bench-marks some of which could be considered as typical programs of a particular artificial intelligence area (such as theorem proving, expert systems, game playing etc.). The focus of the book is, however, on the implementation, and consequently statistics of typical programs can only be inferred indirectly. A wealth of statistics on the allocation of lists by an Interlisp-10 implementation has been collected by Clark and Green.[11] Their focus has been on opportunities for space-efficient encoding, a much more specific concern than ours.

Peyton Jones compares the efficiency of combinator reduction according to Turner to that of applicative order and normal order λ-reduction.[12] The experimental results with a set of small test programs indicate that combinator reduction can be efficiently implemented.

METHODOLOGY

We have run a set of eight existing SASL programs on an interpreter that we had instrumented with a data-collection procedure.

The SASL graph reduction system

Our implementation of SASL closely resembles Turner's implementation.[3] It consists of a compiler to translate a source-language program into an abstract syntax graph and an evaluator which transforms this representation of the program, step by step, into the representation of its result. A representation has the form of a rooted directed binary graph. Interior nodes represent function applications and leaves represent primitive values or functions. The left descendant of an interior node stands for the function and the right descendant for the argument to which it is applied.

The evaluator is capable of applying about three dozen different transformations to a program graph. Each transformation (reduction step) corresponds to the firing of a primitive function. In addition to the 'standard' combinators and arithmetic/logic functions,[3, 13] Turner uses combinators to support pattern matching on arguments of SASL functions (TRY and MATCH) and various optimizations of the standard combinators (e.g. Sp is shorthand for (S' P) and Us is a strict version of U). In its main loop, the evaluator searches the graph leftmost depth-first for a primitive function supplied with a sufficient number of arguments. If such a function is found, the corresponding transformation is applied to the graph and the search continues. Evaluation terminates when no more reducible expressions can be found.

Two issues related to the implementation of storage allocation that Turner could afford to ignore in his paper need to be described here:

1. For all constants of type boolean, character, nil or combinator, *one* node is permanently allocated to represent its value. Such leaves are not included in any of the statistics, because they would bias the values for sharing of nodes.
2. The P combinator is used so often (in list construction) that Turner has introduced a special interior node to be used instead of a Curried application of P to two arguments (see Figure 1). Selecting the *n*th element of a list is executed as *n* successive reduction steps.

Set of application programs

The test set consists of eight different programs. Four programs are small and are run with a small input data set. The four other programs are all of medium size. They are run on small input data sets. The following list provides a short description of each program, the input to which it was applied and the motivation for including it in our test set.

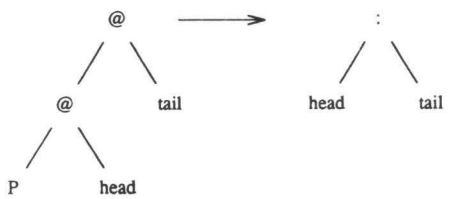


Figure 1. Optimization of storage allocation for list construction. A node marked '@' indicates an application node and ':' a constructor node

fib 7

prints the seventh Fibonacci number using double recursion.

This function appears as an example in most texts on functional programming. Its behaviour during graph reduction may therefore be of general interest.

qsort (9, 0, 1, 8, 4, 8)

sorts a list of 6 numbers using Quicksort.

Apart from being another standard example, this algorithm is typical for the divide and conquer approach to program development, which we expect to become very important for parallel evaluation.

hamming 100

prints in ascending order the first 100 natural numbers whose prime factors are 2, 3 and 5 only.[14]

The interesting aspect of this solution is that the output list is shared among many computations via cycles in the representation. In an implementation that does not support cycles this solution cannot be executed efficiently.

paraff 5

enumerates in order of increasing size the first five paraffin molecules, similar to Turner's original program in KRC.[15]

wave 5

a simplified version of a program that predicts the tides in the North Sea. It assumes a square region with constant depth and simulates five discrete time steps of 20 minutes.

This program is the subject of a study towards developing functional programs that are suitable for parallel evaluation on coarse grain parallel architectures.[16]

em

implements a subset of the commands that are supported by the standard UNIX editor *ed*.

Its input script consists of a call to the help command, three calls to the command to read in a file and a call to the command to print the entire contents of the edit buffer. The input file is the same as that prepared for the *yacc* program (see below). The *em* program has been developed as an exercise in functional programming for a problem area where functional programming is often claimed to be unsatisfactory.[17]

lambda

an implementation of the λ-K calculus,[18] with some added delta rules to support arbitrary-precision arithmetic.

This program was written by one of the authors. The input data consist of the definitions of the standard combinators S and K in the λ-calculus and the application (S K K).

yacc

essentially a rewritten version of the UNIX parser generator of the same name.

It was developed with roughly the same motivation as the *em* program.[19] The input data to the *yacc* program consist of a description of the syntax of input to *yacc* itself. The output produced is a complete parser of *yacc* input, written in SASL.

Table I gives an indication of size and complexity of the eight programs that were selected. The number of lines of pure program text (i.e. excluding comments and blank lines) is a measure of program size. The number of functions defined at global level (i.e. excluding WHERE definitions) is provided as a measure of the program

Table I. Global program characteristics

	fib	*qsort*	*hamming*	*paraff*	*wave*	*em*	*lambda*	*yacc*
Lines of text	3	8	9	71	179	228	560	1250
Global functions	1	2	4	15	33	48	114	163
Bytes read	–	–	–	–	–	2103	61	701
Bytes written	2	6	274	497	3232	2953	918	1622
Reduction steps	232	711	12984	24895	417397	248484	90302	680872
Node claims	289	1101	15970	25453	334956	272524	65312	651707

structure. The total number of (combinator) reduction steps is considered to be representative for the computational complexity of a program. On the average about one node is claimed per reduction step.

Data collection method

The aim of the experiment is to determine the numerical values of a number of parameters that characterize both the process of normal-order graph reduction (e.g. the average number of nodes claimed per reduction step) as well as the intermediate graphs that occur during reduction (e.g. the number of nodes in the graph). For some parameters we are also interested in their fluctuations in time. Although some reduction steps, such as the TRY combinator, take longer to execute than others, we suppose this variation to be unrelated to any of the factors being studied and count time in reduction steps. Each reduction step (i.e. the graph rewrite due to a single combinator) is considered to consume one unit of time. After a certain number of reduction steps (the sampling interval) the evaluator passes control to the data-collection procedure, which traverses the complete graph.

We assume that the sampling is unbiased, i.e. that none of the programs that are studied contains periodic phenomena with a period related to the chosen sampling interval. This implies that the reliability of the statistics we quote is only dependent on the number of samples. The sample intervals chosen for the test programs and the resulting number of samples are shown in Table II.

We have repeated all measurements with a sample interval approximately ten times larger and found only a significant difference in the measurements of the reference count of the current redex (see below).

Table II. Sampling

	fib	*qsort*	*hamming*	*paraff*	*wave*	*em*	*lambda*	*yacc*
Sample interval	1	1	7	11	211	211	67	401
Number of samples	233	712	1855	2264	1979	1178	1348	1698

RESULTS

In the following paragraphs, we present the statistics that we have gathered.

Instruction mix

The majority of the combinators executed by a graph reducer based on Turner's combinators serve to place arguments in the right position. This is a major source of inefficiency,[4, 6] as can be seen from the data in Table III. In the case of the *yacc* program, for instance, no more than 5 per cent of all combinators executed perform 'real work' (HD, TL, list selection, APPEND and the arithmetic and logic operators). Particularly worrying is the popularity of the identity function I. On the positive side is the success of the optimization that uses the B and C combinators and their derivatives instead of the S combinator.

Table III. Percentages of combinators executed

	fib	*qsort*	*hamming*	*paraff*	*wave*	*em*	*lambda*	*yacc*
HD, TL, select	0	2·1	0	0·5	16·3	1·3	2·9	3·2
APPEND	0	1·8	0	0	0	15·8	0·1	0·1
Operators	15·5	2·1	4·9	5·8	12·6	9·7	3·4	1·9
I	17·7	9·9	24·2	23·1	25·8	6·7	47·3	19·7
Y	0	0·7	0	0·8	0·3	0·2	0·7	0·8
K	0	7·3	4·9	6·8	4·4	9·6	4·6	6·4
S, S', Sp	5·2	15·1	15·8	8·1	6·7	0·5	5·9	3·8
B, B', Bp	10·3	34·5	22·3	20·6	16·7	19·8	17·2	42·7
C, C', Cp	10·3	11·4	8·1	10·7	12·4	3·6	6·3	7·6
U, Us	0	5·2	5·2	4·3	3·0	11·5	4·2	8·7
COND	0	2·1	2·6	2·8	0·8	0·2	2·1	1·1
TRY	21·6	3·9	6·0	8·8	0·5	10·6	3·1	1·7
MATCH	19·4	3·9	6·0	7·7	0·5	10·5	2·2	2·3

The size and growth of a graph

The number of nodes in the program graph as a function of time gives an indication of the amount of space required. In particular, if this function fluctuates wildly, the demands on the storage allocator are distributed very unevenly in time. In order to avoid long disruptive pauses caused by the garbage collector, special measures may be considered.[20]

The size of the graph for the medium-sized programs is depicted in Figure 2. The horizontal axis represents normalized time expressed as proportion of the total number of reduction steps for the entire program; the vertical axis shows the number of nodes as a proportion of the maximum attained (shown at top). The results indicate that the total number of nodes in a graph changes rather gradually. All major transitions in the size of the graph can be related easily to the algorithm. The *yacc* and *lambda* programs are similar in the sense that the graph builds up rapidly and then enters a relatively steady state until near the end. The behaviour of the *wave* program (and to a lesser extent that of the *em* program) is interesting because it shows one of the drawbacks of lazy semantics. For more than 95 per cent of its execution time it is building up a large graph that reflects the propagation of the demand to the initial boundary conditions. Most of the real work occurs when this graph is reduced near the end of the computation. We expect this phenomenon to become less pronounced when the simulation is extended over more time steps.

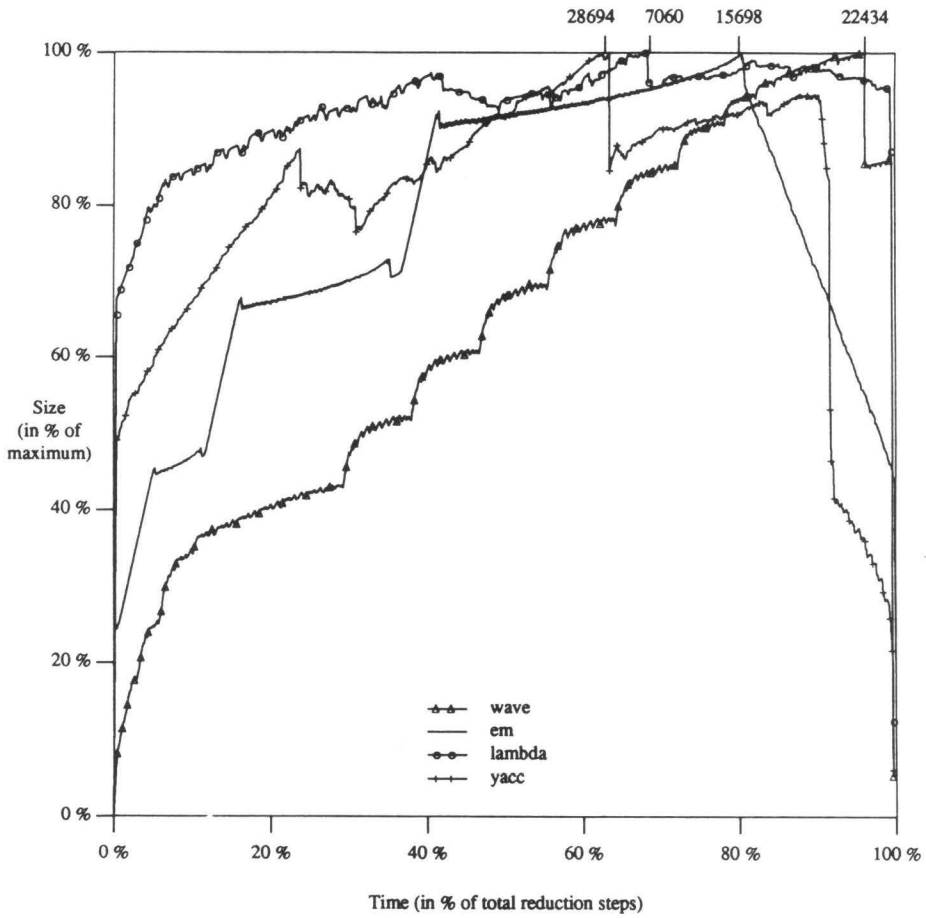


Figure 2. The number of nodes in a graph as a function of time

The shape of a graph

An important parameter for a graph-traversal algorithm is the shape of the average graph. For a recursive descent algorithm, for instance, a balanced graph uses less stack space than a graph of the same size that is severely out of balance. To obtain a measure for this aspect we recorded the depth of each leaf node during leftmost depth-first traversal. The average depth is of course dependent on the size of the graph. We would like to have a more scale-invariant measure. We know that the average depth of planted plane trees is approximately $\sqrt{\pi n}$, where n is the number of nodes.[21] Hence, if our graphs behave like the average tree, the *relative depth* of a node in a graph, defined as

$$\text{relative depth} = \text{depth}/\sqrt{(\text{number of nodes})}$$

would be reasonably scale-invariant. We have recorded the relative depth of each node during each traversal (see Table IV).

Table IV. Ratio between average depth and square root of the number of nodes

	fib	*qsort*	*hamming*	*paraff*	*wave*	*em*	*lambda*	*yacc*
Average	1·6	1·5	2·3	1·3	1·2	0·7	0·9	2·0
95th percentile	2·4	2·4	4·6	2·3	2·0	1·1	1·5	9·4

The composition of a graph

During the evaluation of the eight programs, the composition of their graphs was recorded. In Table V the average percentages of structure nodes are listed. The standard deviation of these figures is less than 10 per cent of the average.

The figures indicate that there is certainly an advantage in the use of constructor nodes. For instance, without the constructor nodes 26 per cent more nodes would have been required for the *em* program. When interpreting these figures, remember that leaves that represent constants of type boolean, character, nil or combinator are not counted as part of the graph.

The structure of a graph

In the combinator code that is evaluated by the SASL interpreter, there are two kinds of simple subgraphs visible that may be replaced by single nodes: constructor nodes chained via their tail fields and application nodes chained via their head fields. The former may be interpreted as the representation of a completely linear list and the latter as the application of a single function to multiple arguments. During the experiments, occurrences of both have been recorded by counting the lengths of the respective chains. Applications of the identity function were ignored. A summary of these figures is shown in Table VI. It shows for instance that for our four medium-sized programs at least 90 per cent of all functions have one or two arguments. In an implementation based on supercombinators these figures would probably be significantly higher. All data structure in SASL programs must be built out of lists. In a language which provides additional data structuring tools, such as records or tuples, fewer short lists would be used than is the case here.

The use of sharing

The major advantage of graph reduction over string reduction is the possibility to exploit sharing. Sharing can be beneficial to avoid duplication of work and duplication of (program or data) storage. The use of sharing has been investigated from a number of different perspectives.

Table V. Average composition (in per cent) of SASL graphs

	fib	*qsort*	*hamming*	*paraff*	*wave*	*em*	*lambda*	*yacc*
Application	78	92	64	80	83	73	86	83
Constructor	–	4	20	17	11	26	11	15

Table VI. Distribution of list lengths and argument counts

	fib	*qsort*	*hamming*	*paraff*	*wave*	*em*	*lambda*	*yacc*
List length								
1 element	–	62%	41%	12%	5%	17%	43%	21%
2 elements	–	2%	0·4%	43%	9%	74%	19%	32%
3 or 4 elements	–	14%	0·9%	33%	8%	0·3%	20%	37%
5–8 elements	–	22%	2%	11%	17%	0·5%	6%	5%
95th percentile	–	5	92	6	12	26	14	8
Argument count								
1 argument	0%	44%	39%	28%	13%	54%	50%	66%
2 arguments	80%	40%	37%	59%	76%	39%	41%	29%
95th percentile	4	3	4	3	3	3	3	2

Reference counting

An important parameter for the storage allocation and reclamation system is the number of pointers to a node. For instance in a system which uses reference counting it is important to know how often the reference count field overflows. The reference count taken over all nodes in all sample graphs has been measured. The average values and 95th percentiles are shown in the first two rows of Table VII. The distribution is such that for the medium-sized programs less than 10 per cent of all nodes are shared (third row). The histograms of reference counts over nodes for the four medium-sized programs are plotted in Figure 3. The horizontal axis represents the reference count and the vertical axis the proportion of the nodes with such a reference count.

During normal-order graph reduction, there is always a current redex. This redex can be identified with an application node in the graph. The bottom half of Table VII shows the sharing figures on the current redex node. Apparently shared nodes have a higher probability of being selected as current node. These figures may not be construed to imply that graph reduction is on the average twice as efficient as string reduction. In string reduction the S combinator, for instance, would copy its argument graph once, but the contractum of the S reduction may appear later as an argument to another S reduction, such that the number of copies rises exponentially. We have run some of our programs in string reduction mode, generally on smaller input data sets, and found the number of reduction steps to be larger by orders of magnitude.

When we repeated the measurements with a larger sample interval, the figures for sharing of the current redex in the *em* program were reduced by a factor of 2. We could not find a explanation for this discrepancy.

Cycles

In a storage allocation system that employs reference counting the presence of cycles in the graph requires special attention.[22] Information about the occurrences of cycles and their length may be useful to delimit the scope of the problem. The data collection algorithm visits the nodes of the graph leftmost depth-first, such that each edge is traversed exactly once. A cycle is counted whenever an edge joins the current path from the root of the graph. The experimental data (Table VIII) show that cycles are

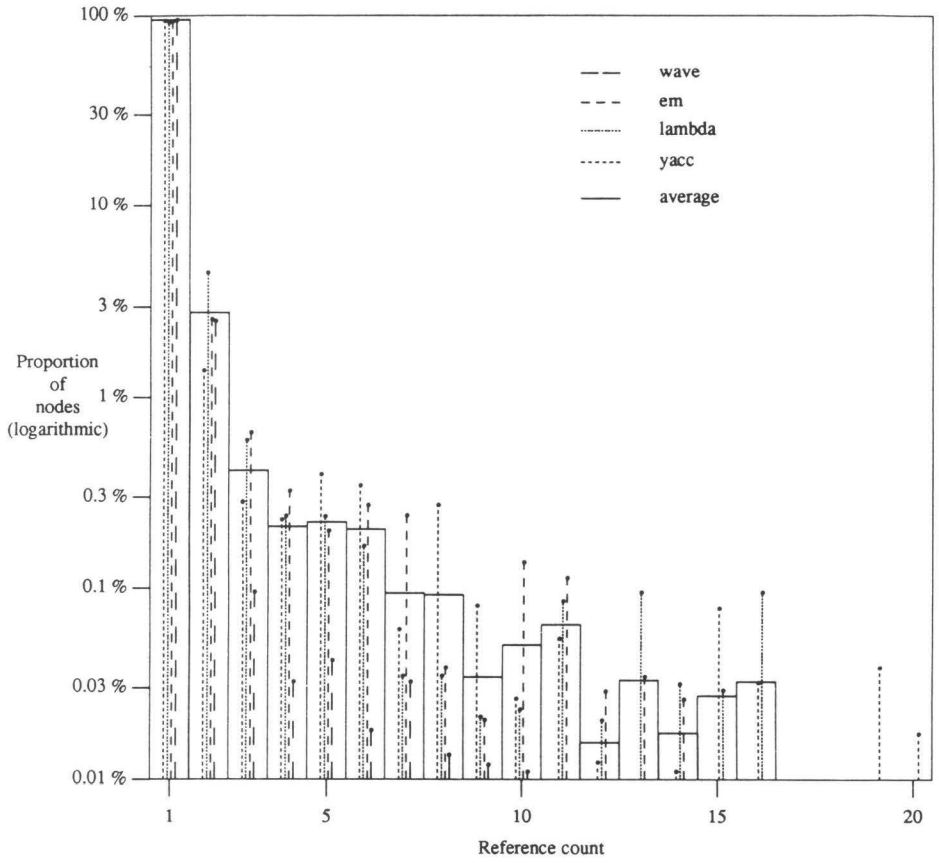


Figure 3. Histograms of the reference counts

small but occur relatively frequently. In the medium-sized programs, there is on the average one cycle per 200 nodes, with an average length of 15 nodes. All programs are recursive, which in this implementation gives rise to cycles (the knot-tying Y combinator). In the *hamming*, *paraff* and *wave* programs cyclic data structures also occur.

Ongoing research at our institute is investigating the question of how cycles can be avoided.[23] As an illustration we present some preliminary results in the lower half of Table VIII. These were produced by replacing the knot-tying Y combinator in our reducer by one that avoids cycles. This leads to a large increase in the number of reduction steps. To limit this increase we modified seven of our SASL programs by lifting all recursive WHERE clauses to global level and removing WHERE clauses with multiple definitions. Recursive global functions still require pointers back to their roots, but these were threaded through an indirection table and not counted as cyclic.

Table VII. Reference counts

	fib	*qsort*	*hamming*	*paraff*	*wave*	*em*	*lambda*	*yacc*
All nodes								
average	1·17	1·13	1·13	1·21	1·50	1·17	1·14	1·16
95th percentile	2	2	2	2	1	1	1	2
percentage shared	13	8	11	14	4	3	5	6
Current redex								
average	1·9	1·3	1·5	2·1	2·1	2·9	1·7	2·1
95th percentile	3	3	3	5	4	22	3	3
percentage shared	56	19	37	47	36	20	56	34

Table VIII. Cycles

	fib	*qsort*	*hamming*	*paraff*	*wave*	*em*	*lambda*	*yacc*
Standard cyclic implementation								
cycles per 1000 nodes	30·1	23·1	17·5	9·3	0·7	2·3	10·9	4·8
average length	4·8	7·0	15·4	13·2	17·1	7·8	12·9	22·7
Cycle avoiding experiment								
cycles per 1000 nodes	0	0	3·2	0·0009	0·002	0	0	not run
average length	0	0	54·4	31	6	0	0	not run
extra reduction steps	0%	0%	0%	0·16%	7·4%	7·3%	9·8%	not run

The life span of nodes

The life of a node starts when it is claimed by the evaluator and ends when it becomes unreachable from the root. Information about the life span of nodes and its distribution could be used for instance to design a hierarchical node store, where the residence of a node is determined by its retention period, as exploited by Ungar.[20]

In our experiments we determined an upper bound for the life expectancy of nodes: the creation time of each node is known exactly, but the expiration of nodes that are part of a cycle is assumed to have occurred at the time at which the data collection algorithm discovered the node to be garbage (Table IX). Because cycles are rare this lack of precision is only slight.

From the experiments it was found that most nodes have a short life: about 60 per cent of the nodes witness no more than 10 reduction steps. The histograms obtained for the medium-sized programs are shown in Figure 4. The horizontal axis represents lifetime and the vertical axis the proportion of the nodes with a lifetime around that value.

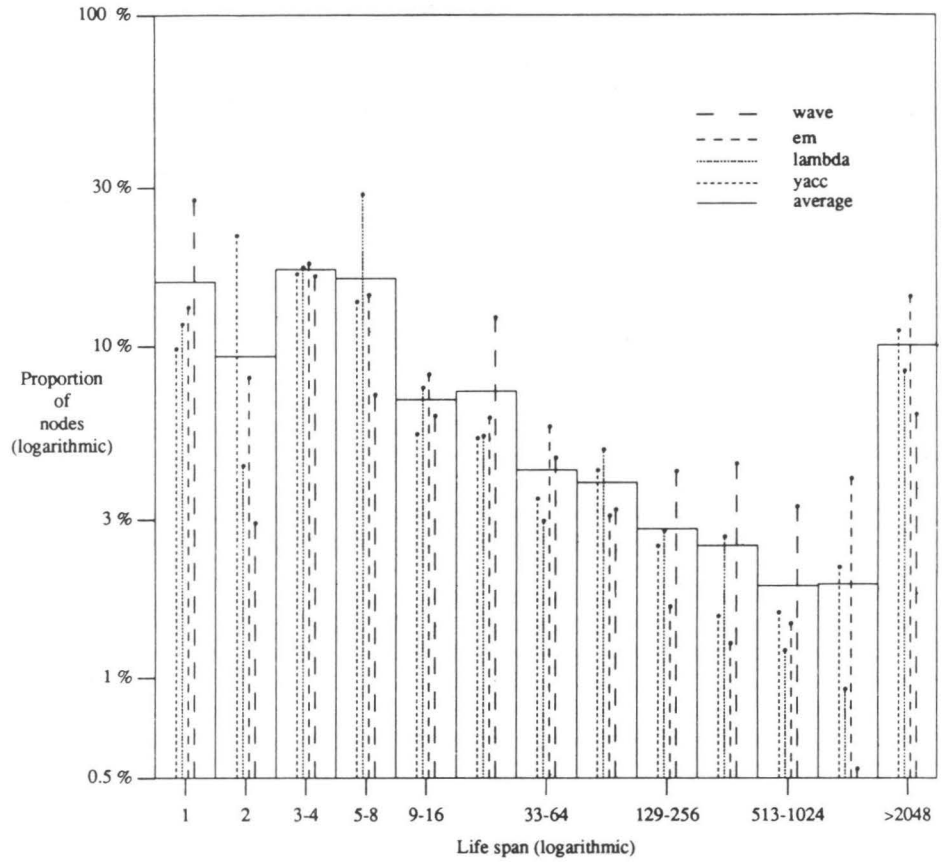


Figure 4. Histograms of the life time of nodes in reduction steps

Table IX. Life span of nodes measured in reduction steps (per cent)

	fib	*qsort*	*hamming*	*paraff*	*wave*	*em*	*lambda*	*yacc*
1 step	1	15	21	15	10	28	13	12
2–10 steps	68	48	43	41	55	29	44	54
11–100 steps	20	18	25	19	15	23	19	16
100–2500 steps	11	19	9	17	10	14	10	10

CONCLUSIONS

The most notable property of the graphs we have measured is their lack of distinction. They do not have disproportionally long paths, but are not very balanced either. The graphs do not grow or shrink in sudden bursts. In most aspects their structure as well as their behaviour is quite ordinary. This is fortunate, because it means that the behaviour of a combinator-based graph reducer can often be approximated by a very simple model. An example of this can be found in Table IV, where the depth of the average tree was used to approximate the depth of nodes of a graph.

For most programs the graph grows in an initial phase to a certain size around which it fluctuates for the major part of the computation and which is then reduced to the result in the final phase. Usually the central phase accounts for more than 80 per cent of the reductions, during which the size fluctuations are less than 20 per cent. In the central phase less than 10 per cent of the nodes are shared (i.e. have a reference count greater than one). This leads to the conclusion that, as in SKIM II, the one-bit reference counting technique[24] (with the count stored in the pointer) could have been used successfully in our SASL implementation. This implementation produces on the average one small cycle in a few hundred nodes. At the cost of some extra reduction steps the number of cycles can be reduced by orders of magnitude. About one node is claimed and another one released per reduction step. Most nodes only live a short time.

Table V shows that more than 94 per cent of the nodes in the graphs of the four medium-sized programs represent structure, not pure data values. If this is also true for really large programs, a representation in which much more structure is encoded implicitly could yield tremendous savings, both in time and in storage. Reduction systems based on super or serial combinators are steps in this direction.[4–6] The optimization by the special constructor nodes could be extended to include nodes for other data structures, such as arrays and records. The disadvantage of such an approach is that the storage allocation and reclamation system will have to be able to cope with nodes of arbitrary size. The compensation for the extra effort to support nodes of arbitrary size must come from savings in space and/or time. However, the results in Table VI indicate that long lists are relatively rare. The introduction of cells of arbitrary size in the SASL implementation would therefore not be worth while. Instead, an additional type of node with, for instance, a maximum of four pointers could be considered. These results should be interpreted with care. The SASL implementation is fully lazy and evaluation is driven by the need to print. Therefore, unless a list is shared, most of the time only a small part of it will be present. In an implementation that supports eagerly-evaluated data structures the situation is very different.

ACKNOWLEDGEMENTS

Some of the ideas worked out in this paper arose in discussions with Betsy Pepels. Wim Vree and two anonymous referees made valuable comments on draft versions of the paper. We used the compiler developed by Riet Oolman. Her help and that of Henk Keller during the development of the instrumentation software were greatly appreciated. Lex Augusteijn and Theo van der Lee translated the paraffin program from KRC into SASL. This work is sponsored by the Dutch ministry of Science and Education, dienst Wetenschapsbeleid.

REFERENCES

1. H. P. Barendregt, M. C. J. D. van Eekelen, M. J. Plasmeijer, P. H. Hartel, L. O. Hertzberger and W. G. Vree, 'The Dutch parallel reduction machine project', *Proceedings of the International Conference on Frontiers in Computing*, Amsterdam, North Holland, December 1987.
2. A. H. Veen, 'Analytic modeling of a parallel graph reducer', *Technical Report 79*, Sectie informatica, University of Nijmegen, 1986.
3. D. A. Turner, 'A new implementation technique for applicative languages', *Software—Practice and Experience*, **9**, (1), 31–49 (1979).
4. R. J. M. Hughes, 'Super combinators — a new implementation method for applicative languages', *Proceedings of the ACM Symposium on Lisp and Functional Programming*, Pittsburgh, 1982.
5. P. Hudak and B. Goldberg, 'Serial combinators: "optimal" grains of parallelism', in *Functional Programming Languages and Computer Architectures*, Springer Verlag, Berlin, 1985.
6. T. Johnsson, 'Efficient compilation of lazy evaluation', *SIGPLAN Notices*, **19**, (6), 58–69 (1984).
7. T. Hikita, 'On the average size of Turner's translation to combinator programs', *Journal of Information Processing*, **7**, (3), 164–169 (1984).
8. J. C. Mulder, 'Complexity of combinatory code', *Technical report no. 389*, Department of Mathematics, University of Utrecht, August 1985.
9. R. P. Weicker, 'Dhrystone: a synthetic systems programming benchmark', *Communications of the ACM*, **27**, (10), 1013–1030 (1984).
10. R. P. Gabriel, *Performance and Evaluation of Lisp Systems*, MIT Press, Cambridge, 1985.
11. D. W. Clark and S. C. Green, 'An empirical study of list structure in LISP', *Communications of the ACM*, **20**, (2), 78–87 (1977).
12. S. L. Peyton Jones, 'An investigation of the relative efficiency of combinators and lambda expressions', *Proceedings of the ACM Symposium on LISP and Functional Programming*, Pittsburgh, 1982.
13. D. A. Turner, 'Another algorithm for bracket abstraction', *Journal of Symbolic Logic*, **44**, (2), 267–270 (1979).
14. D. A. Turner, 'Functional programs as executable specifications', *Phil. Trans. R. Soc. Lond*, **A 312**, 368–388 (1984).
15. D. A. Turner, 'The semantic elegance of applicative languages', *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, Portsmouth, New Hampshire, 1981, pp. 85–92.
16. W. G. Vree, 'The grain size of parallel computations in a functional program', *Proceedings of the International Conference on Parallel Processing and Applications*, L'Aquila, Italy, September 1987, Elsevier Science Publishing.
17. P. W. M. Koopman, 'Interactive programs in a functional language: a functional implementation of an editor', *Software—Practice and Experience*, **17**, (9), 609–622 (1987).
18. H. Glaser, C. Hankin and D. Till, *Principles of Functional Programming*, Prentice-Hall, Englewood Cliffs, New Jersey, 1984.
19. S. L. Peyton Jones, 'Yacc in SASL — an exercise in functional programming', *Software—Practice and Experience*, **15**, (8), 807–820 (1985).
20. D. Ungar, 'Generation scavenging: a non-disruptive high performance storage reclamation algorithm', *ACM Software Engineering Notes/SIGPLAN Notices on Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, 1984, pp. 157–167.
21. N. G. de Bruijn, D. E. Knuth and S. O. Rice, 'The average height of planted plane trees', in R. C. Read (ed.), *Graph Theory and Computing*, Academic Press, 1972.
22. D. R. Brownbridge, 'Cyclic reference counting for combinator machines', in *Functional Programming Languages and Computer Architectures*, Springer Verlag, Berlin, 1985.
23. W. G. Vree, 'On the need of cycles in graph reduction', *Internal report D–16*, Computing Science Department, University of Amsterdam, 1987.
24. W. R. Stoye, T. J. W. Clarke and A. C. Norman, 'Some practical methods for rapid combinator reduction', *Proceedings of the ACM Symposium on Lisp and Functional Programming*, Austin, Texas, 1984, pp. 159–166.

A comparative study of three garbage collection algorithms

Pieter H. Hartel

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

Abstract

The running cost of garbage collection is studied as a function of the amount of available store. A performance model originally proposed by Hoare is modified to support experiments with three garbage collection methods: reference count, mark/scan and two-space copy. By also taking the boundary effects into account we show that adding more store to a list processing system with a non-reference counting garbage collector does not necessarily make it run faster. We present an explanation for this anomaly in the behaviour of garbage collection algorithms.

Key words: garbage collection performance modelling combinator graph reduction anomalous behaviour subinstruction level timing

1. Introduction

The implementation of a list processing system requires a garbage collector if storage space is limited and the removal of the last reference to an object is implicit. The major problem is to control the cost at which the act of removing the last reference to an object can be detected. Many algorithms have been proposed to implement garbage collection.[1] Each algorithm has specific advantages and disadvantages. For instance reclamation of the space occupied by cyclic structures is more difficult with reference counting algorithms than with mark/scan or two-space copying algorithms.[2,3] The space requirements and running costs of the algorithms are also vastly different. Based on certain assumptions, mainly about the structure of the graphs involved, theoretical work provides insight in the asymptotic complexity of garbage collection algorithms. A systems architect wishing to select an appropriate garbage collection algorithm for a particular list processing system in the first place chooses the algorithms with the best asymptotic complexity that fit the boundary conditions the list processing system imposes. The practical implementation of these algorithms (all with the same asymptotic complexity) may turn out to be rather different because of the constants that have been ignored in the asymptotic analysis. However, it is precisely this difference that allows the architect to decide which of these algorithms to use. Therefore some

authors have made a more practical analysis of garbage collection, usually based on an implementation in a (hypothetical) language. This makes it possible to base more detailed comparisons on counting memory references, executed instructions etc.[4,5] In this paper the analysis and measurement of the cost of garbage collection is based on subinstruction level timing. This provides the most detailed means of comparing (implementations of) algorithms. This approach was chosen to test the frequently made assertion that counting instructions at a particular level leads to the same conclusions as measuring execution time.

Our analysis is also different from most of the studies that have been published in the literature, because the structure of the graphs we perform garbage collection on is specific to applications of a list processing system. This allows us to derive realistic performance data on garbage collection algorithms. The alternative is to use arbitrary graphs, such as random trees, left degenerate trees etc. We have serious objections against this practice because it seems to be inspired more by the concern that the graphs must be easy to generate than a real motivation of why particular structures should occur in real list processing systems.

The choice for using real graphs directs us towards using a generator of real graphs and garbage. We have chosen to use as a list processing system an implementation of the functional programming language SASL,[6] together with a benchmark of application programs written in SASL.[7] In selecting the application programs we have tried to avoid the unusual ones and thus create a sample of application programs, that represents average behaviour. The benchmark set contains four small programs and four medium size programs. All are run on small input data sets: "*fib* 7" prints the seventh Fibonacci number, "*quicksort* (9, 0, 1, 8, 4, 8)" sorts a list of 6 numbers, "*hamming* 100" calculates in ascending order the first 100 natural numbers whose prime factors are 2, 3 and 5,[8] "*paraff* 5" enumerates in order of increasing size the first five paraffin molecules,[9] "*wave* 5" predicts the tides in a rectangular estuary of the North Sea over a period of 5×20 minutes,[10] "*em script*" runs a simple script through a functional implementation of the UNIX text editor,[11] "*lambda* (*S K K*)" evaluates to *I* on an implementation of the λ-K calculus[12] and "*yacc yacc –grammar*" generates a parser for yacc input in SASL.[13]

The implementation of SASL is based on Turner's method of graph reduction, which uses a fixed set of combinators.[14] As an implementation technique of a functional programming language with normal order semantics, this method has been surpassed by more efficient methods such as the G-machine,[15] CLEAN,[16] and the TIM machine.[17] These methods in essence derive their improved efficiency from avoiding to use the graph whenever possible. Avoiding to use the graph implies avoiding garbage collection, which is contrary to our intention to study garbage collection. There are even applications that do not need the graph at all. Admittedly such programs (e.g. *fib*) do

not represent the most interesting list processing applications, but unfortunately they are often used in comparing the performance of implementations of functional languages. Because Turner's reduction methods places the heaviest demand on the garbage collector we consider it much more suitable as a generator of graphs (and garbage) than any of the more advanced implementation methods of functional programming languages. Turner's method has the added advantage that it is much simpler to reason about than other methods. We know for instance that a reduction step never requires more than three new nodes. This is important to know, because during a reduction step the graph need not be connected. This is the case if a new internal node is added to the graph, because that requires updating at least two pointers: one to the new node and one emanating from it. Garbage collection algorithms would cause disaster if applied to a graph in disconnected state. With program derived super combinators a threshold on the required number of available nodes is application dependent and therefore more difficult to use than with Turner's method.

2. Selection of garbage collection algorithms

We have chosen three garbage collection algorithms that seemed most promising with respect to their run time efficiency. This means that we are prepared to expend a number of extra bits or bytes per object to support rapid storage allocation and reclamation. The mutator that we use is always the implementation of SASL based of fixed combinator graph reduction. It is assumed that both the mutator and the collector run on the same processing element, which consists of a sequential processor and a fair amount of local store. The mutator and the collector run in an interleaved fashion. Normally the mutator executes, but whenever necessary, the mutator calls the collector, which after completing its task returns control to the mutator. Hence we are not concerned with issues of parallel garbage collection. Another problem that is ignored here is the use of virtual memory.

Two more technical problems for which many intricate solutions have been proposed are resolved in favour of simplicity. Firstly, all objects are assumed to occupy the same amount of space (we make no distinction between an object and the space it occupies). Secondly, reclamation of the space occupied by redundant cyclic structures is considered unnecessary. These restrictions poses no problems to the SASL implementation or the benchmark programs. An interior node of the graph that is processed by the SASL interpreter represents the application of a function to a single argument. Leaf nodes represent constants such as (floating point) numbers and combinators. A small object that can accommodate two pointers or a floating point datum suffices. The second restriction is justified because we have observed that the vast majority of the cycles are due to the use of recursive functions and not cyclic data structures.[7] The graphical structure that represents the function definitions of a program is rarely a

good candidate for being garbage collected. Though some of the function definitions may not be used any more after the application has progressed to a certain point, it appears that in particular because of lazy evaluation few functions definitions can be garbage collected. Furthermore the implementation of a functional language is often embedded in a conversational environment that requires the function definitions to be protected from the garbage collector. Instead it is the rapidly expanding structure that represents applications of the user defined functions to their arguments that needs to be controlled by the garbage collector. Cycles in the structure that represents the function definitions can be made detectable at low cost.[18] The remaining cycles are in the data structures. Vree has shown that such cycles can often be avoided by suitable program transformation.[19]

There are in principle two ways to identify redundant objects. The first method, reference counting, keeps a count of the number of references to an object. When the count drops to zero, the object can be recuperated. This method has a strong intuitive appeal, because it "never leaves to tomorrow what can be done today". The disadvantage of reference counting lies in the additional information that must be maintained for each creation or destruction of a reference. The particular algorithm in this category that we choose to study is referred to as *RefCount*. It uses a reference count field that occupies the same number of bytes as a pointer field, such that a reference count can not overflow. *RefCount* always reclaims the maximum number of objects.

The second method to identify redundant objects is by elimination. The objects that are still in use are marked, such that unmarked objects are deemed to be garbage. We selected two algorithms from this category. A mark/scan algorithm first marks all the used objects, then makes a scan over the entire store, picking up all redundant objects. The particular algorithm we chose, referred to as *MarkScan*, delays the scan until objects are actually required. This obviates the need for building a free list and taking it apart again, which would increase running time. The third algorithm (*Queue*) copies the objects that are still in use from the current partition, which will be called *from-space* to a fresh area of store (called *to-space*). The roles of *from-space* and *to-space* are interchanged after each *Queue* garbage collect.[20]

3. System model

We will develop an analytic model to capture the essence of the three different garbage collection methods in a number of parameters. The parameters will be chosen such that the three algorithms can be compared. The behaviour of the algorithms is primarily determined by the size of the store and the number of objects that an application of the list processing system requires to execute. Let the size of the store be S objects and let N represent the number of objects claimed during the execution of a

particular application program. We do not consider loading the application into the store as part of the execution becaused the loading process does not require services of the garbage collector. The number of objects claimed to load an application is represented by the parameter B. The total number of objects ever required by an application program is therefore $B+N$, but the cost analysis pertains to the N objects claimed during execution.

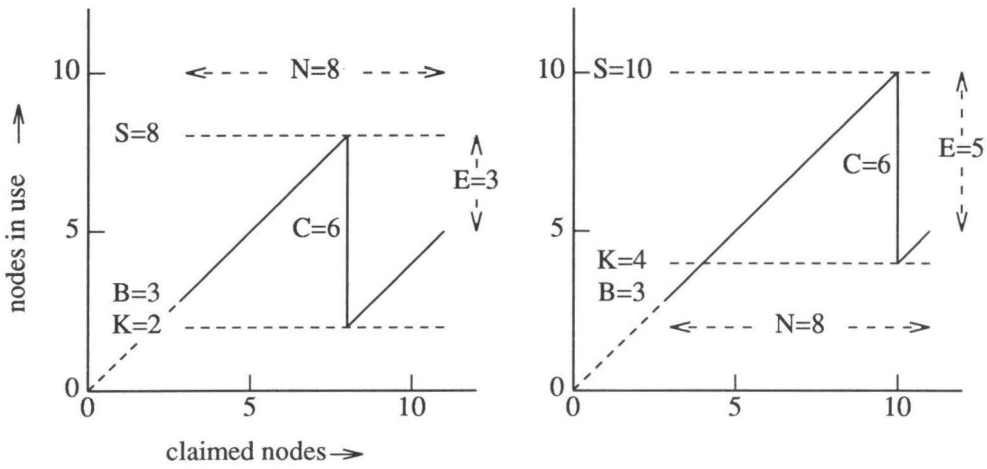
If based on reference counting, the collector must be called for each reference that is deleted, but the *MarkScan* and *Queue* algorithms require the services of the collector only when a certain threshold on the use of the store is reached. As a consequence the behaviour of these systems is rather different from the reference counting scheme. A reference counting algorithm makes objects available to be claimed again at the moment they become unreachable. The cost of reference counting is therefore independent of S. We will define as the cost of reference counting garbage collection the total number of ticks spent in the allocation of new objects and collection of redundant objects divided by N. We use a "tick" as the unit cost of work. When the implementations of the garbage collection algorithms are discussed we will introduce a more concrete measure to use instead of the tick.

The cost of storage management with *MarkScan* and *Queue* does depend on S hence the analysis will have to be more involved. The behaviour of *MarkScan* and *Queue* can be shown in a "storage profile" that represents the number of objects that are in use as a function of the number of claims. We call an object "in use" if it is either part of the graph or when it has become garbage without having been discovered as such. Objects that are in use can therefore not be claimed until the next time garbage is collected. Figure (1) shows the storage profiles of a program with two different values of S (8 and 10 respectively). After loading the program in store, which requires $B = 3$ objects, the number of claims rises until the store is full. As a result of garbage collection a number of objects are recuperated. After garbage collection all objects that are still in use are part of the graph.

We use K to represent the average size of the graph immediately after a *MarkScan* or *Queue* garbage collection. The number of garbage collections is G and C indicates the total number of objects recuperated by *MarkScan* or *Queue* garbage collections. In figure (1-a) $K = 2$ and in (1-b) $K = 4$. In both cases $C = 6$ and $G = 1$.

When an application terminates a number of objects remain available to be claimed that are not needed any more. Let E represent this number of objects The number of objects that are in use when the application terminates is thus $S-E$. In figure (1) the values of E are 3 and 5. In discovering the E objects that remain available at termination a number of ticks are spent by the garbage collector that must taken into account as part of the running costs of garbage collection, in particular if N is a little larger than S. Hoare's analysis,[21] upon which the current one is based, ignores these

boundary effects, since he assumes that S is small in comparison to N. (The second major difference with Hoare's analysis is that we do not account for the size of the store as a cost factor.) In the next section we will show that it is important to include boundary effects in the cost analysis of garbage collection.



(a) $S = 8$; *cost* $= 8\beta + 2\alpha$ ticks (b) $S = 10$; *cost* $= 8\beta + 4\alpha$ ticks

Figure 1 : Two storage profiles of the same application with the *Queue* garbage collector

The parameters (all upper case letters) that we have introduced so far characterise the storage profile of an application. They represent a number of objects in a particular state. What we need to add is a number of cost factors, that can be multiplied by a one of the parameters to calculate the total cost involved in processing objects in that particular state. We will use greek letters to represent cost factors. The most interesting cost factor is α: the average number of ticks spent to traverse an object during the mark and copy phase of *MarkScan* respectively *Queue* garbage collection. Included in the cost α are the ticks to discover whether the object being traversed is a leaf or an interior node and the ticks necessary to mark the object as being traversed (*MarkScan*) or to mark it and copy the object to *to-space* (*Queue*). The value of α is thus sensitive to the composition of the graph: the more objects are shared, the higher α will be. The second cost factor β represents the number of ticks required to claim an object. Both for *MarkScan* and *Queue* garbage collection β is a constant, because it does not depend on the structure of the graph that garbage collection operates on. The value of β for a particular implementation of a garbage collection algorithm can be determined statically from the program text; that of α must be measured.

The behaviour of the *Queue* algorithm can be fully described by the parameters S, N, B, E, K, C, G and the cost factors α and β. For the *MarkScan* algorithm we need one

more parameter and an associated cost factor, because when claiming an object, we must skip (and unmark) the marked objects that we find on our way to an unmarked object. The parameter A represents the total number of marked objects that must be skipped and the cost of skipping a single object is represented by σ. Like β the factor σ can be determined statically from the program text of the implementation of the garbage collection algorithm. In general A can not be determined from the other parameters because its value depends on the way marked objects are distributed over the store. The only case where A can be computed is when $E = 0$ (then $A = G\ K$).

parameter	meaning
	applicable to all three algorithms
B	total number of objects in use when execution Begins
N	total Number of objects claimed during evaluation
	applicable to *Queue* and *MarkScan*
E	total number of unclaimed, free objects when the application Ends
C	total number of Collected objects
S	total number of objects the Store may hold.
K	average (over collections) number of objects in use after garbage collection
μ	ratio S / K
G	number of Garbage collections
α	average (over a run) number of ticks to mark or copy an object
β	number of ticks to claim an object
	applicable to *MarkScan* only
A	total number of skipped objects during the scAn
σ	number of ticks to skip a marked object

Table 1 : Model parameters.

The meaning of all parameters and cost factors that we have introduced is summarised in table (1). The names of the relevant parameters and their values with respect to the execution of a sample program are also shown in figure (1).

3.1. Anomalous behaviour in garbage collection

Garbage collection performance at values of $S \leq N \leq 2S$ is interesting because it represents the performance of a system with garbage collection performed under favourable conditions. (The case where $N < S$ does not require garbage collection and is therefore not considered here). These favourable conditions are not only of

theoretical interest because stores that are large enough to sustain a significant amount of computation without requiring more than one garbage collection are not uncommon. However when N is just a little larger than S, the garbage collector has to perform much work to recuperate redundant objects while few of these will actually be claimed. The average cost per object will therefore be relatively high. One would expect this cost to drop as the size of the store is enlarged. We will show that this is not necessarily the case.

Figure (1) shows two storage profiles of the same application with the *Queue* garbage collector. In both cases B and N have the same values, but the sizes of the stores are different. Since in this example $G = 1$, we find for the total cost of allocation and garbage collection $N\ \beta + K\ \alpha$. As we can see from the diagram, the value of K rises along with that of S from 2 to 4. We also know that β is a constant of the implementation of the garbage collection algorithm and that N is a constant determined by the application. The α values may be different because they are sensitive to the structure of the graph being copied. However, a difference of a factor two is virtually impossible because that would imply that in the graph traversed in figure (1-a) many more nodes would be shared than in figure (1-b). In practice we have observed that on the average the percentage of shared nodes is less than 10%.[7] The increase of K can not be compensated by a decrease of α, hence the total cost of allocation and collection is higher in (1-b) than it is in (1-a). We must conclude that adding more store does not always make a list processing system faster. The real problem is that there is no easy way to choose a favourable moment for garbage collection. The only way to know when a graph is small is by performing garbage collection....

The effects of the "garbage collection anomaly" are not restricted to the case where $G = 1$, but the effect at higher G values has a tendency to average out. In the experiments discussed in section 5 we will show the effect of garbage collection anomaly on a real application.

3.2. Analysis of *Queue* and *MarkScan* model parameters

Not all the parameters for the *Queue* and *MarkScan* algorithms in table (1) are independent. We will show that G and C can be eliminated. The first observation we can make, is that an object must either be in existence when the application is started (B), or claimed (N), or still available upon termination (E). Their sum must be equal to the total amount of space before the first garbage collection is started (S) plus the number of collected objects (C):

$$S + C = B + N + E \tag{1}$$

The second observation is that garbage collection can not alter the number of objects

in the system. The number of collected objects can be expressed in two ways, which must both yield the same result:

$$G\ (S - K) = C \tag{2}$$

Combination of (1) and (2) yields:

$$G = \frac{N + B + E - S}{S - K} \geq 0 \tag{3}$$

The total cost of marking or copying the objects that are part of the graph is $G\ K\ \alpha$. For the *Queue* algorithm, the total cost of claiming objects is $N\ \beta$. The average number of ticks necessary to claim an object for the *Queue* algorithm is therefore:

$$ticks\ per\ claimed\ object_{Queue} = \frac{N\ \beta + G\ K\ \alpha}{N}$$

The case of *MarkScan* is slightly complicated by the delay of the scan. While the cost of claiming an object is a constant for *Queue*, *MarkScan* when claiming an object must skip the marked objects on its way to an unmarked object. This requires a total of $A\ \sigma$ ticks. For the *MarkScan* algorithm we find as the average cost per object:

$$ticks\ per\ claimed\ object = \frac{N\ \beta + G\ K\ \alpha + A\ \sigma}{N} \tag{4}$$

To facilitate references to the formulae we will use (4) also for the *Queue* algorithm, but with $\sigma_{Queue} \equiv 0$.

To compare the costs incurred by the different application programs, we will express the cost of allocation as a function of normalised store size $\mu = S\ /\ K$. With the value of G found in (3) we obtain from (4):

$$ticks\ per\ claimed\ object = \beta + \frac{N + B + E - \mu K}{(\mu - 1)\ N}\ \alpha + \frac{A}{N}\ \sigma \tag{5}$$

This formula will be used to calculate α from the other quantities, that we will determine in a series of experiments to be described below.

4. Implementation of garbage collection algorithms

The SASL interpreter, which drives the storage allocation and reclamation, operates on a rooted directed cyclic binary graph. The graph is transformed from its initial state to the final state in small "reduction" steps. On the average a step claims one node and alters a few edges. No reduction step requires more than three new nodes. With *Queue* or *MarkScan* garbage collection we use this property to check at the beginning of each step, if enough objects are still available. If not, the garbage collector is started with

the root of the graph as the first object to be marked or copied. At that point, the graph is connected, which is usually not the case while a reduction step is in progress. Because of its close connection with reduction we have decided not to account for the test on the available space as part of the cost of allocation and collection.

Apart from the pointer to the root, the SASL interpreter maintains an stack of pointers into the graph in its "left ancestors" stack. The average depth of this stack is of the order $\sqrt{K}$.[7] The contents of the left ancestors stack has to be updated by the *Queue* algorithm and the cost of these updates is accounted for. The left ancestors stack is of no concern to the *MarkScan* and *RefCount* algorithms.

The representation of a node includes a tag field that allows the reducer and the collector to perform case analysis on the type of the node. The collector must be able to distinguish leaf nodes from interior nodes but the reducer needs more information. Since our interest is in the cost of garbage collection rather than the cost of reduction we have decided to account for reading and writing pointer fields only. At all times we assume these fields to contain valid information (i.e. a pointer value or NIL). This may appear to give the *Queue* algorithm an advantage, since insufficient data is copied, but as we shall see in section 5.1, the difference may safely be ignored.

All algorithms would suffer in roughly the same way if discrimination on the tag value had been incorporated in the performance measurements, but there exist also methods that do not require tags to distinguish between leaf and interior nodes. For instance leaf nodes could be kept in a separate region of store such that the pointer values carry the type information. Our objects can thus be considered as minimal objects.

Table (2) shows the composition of the objects that are processed by the algorithms (as far as storage allocation and reclamation is concerned). A pointer or a reference count field requires 4 bytes and a flag field requires 1 byte of storage. Mappings that are thriftier of space are conceivable, but on most architectures this would slow down execution.

Algorithm	Extra field(s)	maximum space in bytes for			MC68010 cycles	
	per object	Heap	Stack	Total	β	σ
RefCount	reference count	12 S	4 S	16 S	-	-
Queue		2×8 S	0	16 S	22	-
MarkScan	visited flag	9 S	4 S	13 S	36	50

Table 2 : Static characteristics of the algorithms on the MC68010 processor

The garbage collection algorithms were implemented in assembly language for a Motorola 68010 processor. In the measurements a processor clock cycle ($\approx$ 0.1 μ sec)

is taken as the unit of time (*tick*). Recursion is implemented via explicit use of a stack of pointers to the objects that remain to be processed. A garbage collector operates in a tight loop consisting of move, clear, add, increment, decrement, test and branch instructions. Often used data is kept in 8 of the processors registers. None of the algorithms could be improved by using a few more registers. The flow of control has been structured such that at each decision point the condition that occurs the least frequently causes the branch to be taken. No procedure calls are used; the code to claim a new node is inserted as "in line" code where needed. Since most information is kept in registers care was taken to use the settings of the condition codes generated by "move to register" instructions rather than test instructions to steer the branch instructions. The fields of an object are accessed by using the "address register indirect with displacement" addressing mode and the stack is manipulated with "auto increment" and "auto decrement" addressing modes. The configuration of a processing element is assumed to be such that the instructions, the data in the stack and in the heap can each be accessed at uniform cost, so we assume no memory management unit to be interposed between the processor and the store. The effects of caches on the performance of the algorithms are included by simulation in our experiments.

The implementations of the algorithms are close to what may be considered "standard",[1] such that we need not provide the listings here. The information that has been determined from the program text of the implementations are the cost factors β and σ. Their values expressed in MC68010 processor cycles are shown in the last two columns of table (2). We use a memory that does not require "wait states" hence one memory reference requires four processor cycles.[22]

5. Experiments

Each application from the SASL benchmark set has been run once with *RefCount* and about ten times with each of the other garbage collection algorithms. The parameter S was varied from run to run in order to measure the dependency of the *cycles per claimed object* on μ with the *MarkScan* and *Queue* garbage collectors. The values of S close to K cause many garbage collections, whereas large values of μ cause few or no garbage collections. Equation (5) states that *cycles per claimed object* is roughly proportional to the reciprocal of μ. This is confirmed by the measurements. As an example figure (2) shows the experimental data that have been gathered with the *lambda* program.

The cost of *RefCount* does not depend on μ and is about an order of magnitude higher than any of the β values. Hence for each non-reference counting algorithm, there is a cross over point, where *RefCount* has the same cost as non-reference counting. *RefCount* is cheaper below this point.

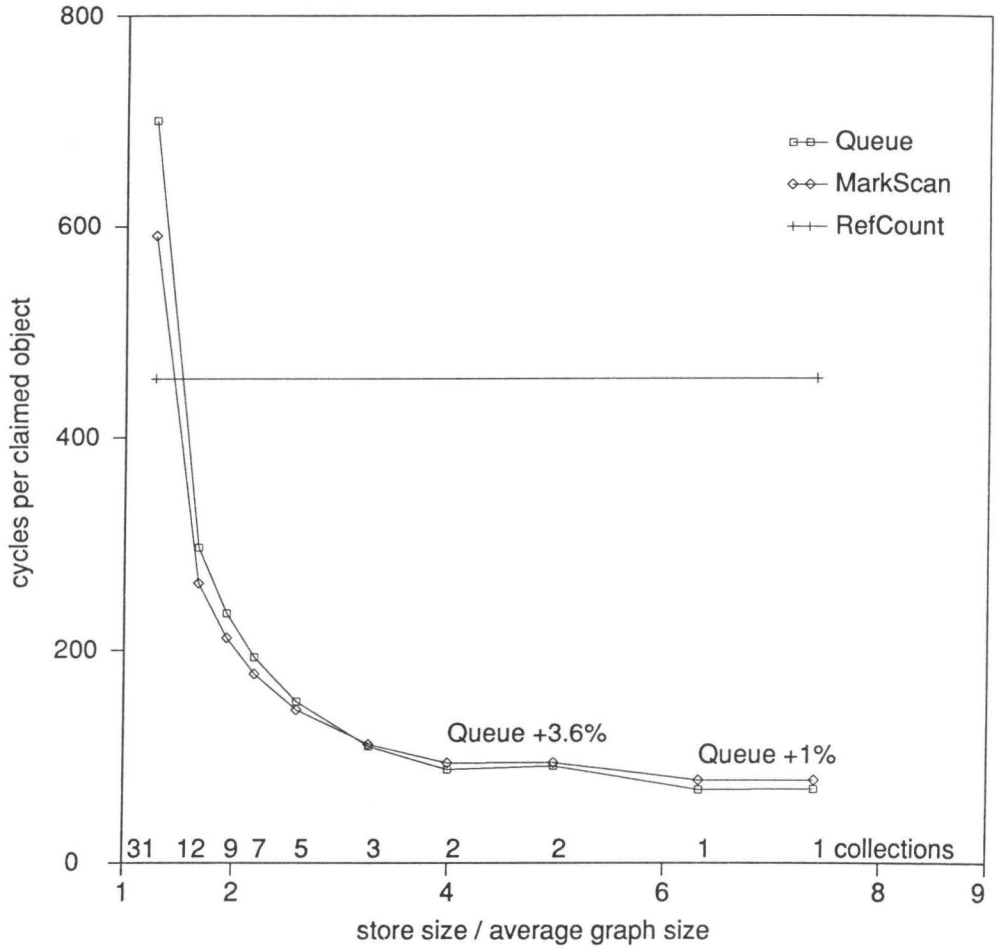


Figure 2 : Measured *cycles per claimed object* in MC68010 processor cycles with *lambda*

From the rise of the *cycles per claimed object* between $\mu = 4$ and 5 and again between $\mu = 6.3$ and 7.4 we can see that in a real application the performance of the *Queue* algorithm suffers noticeably from the garbage collection anomaly. The increase in the *cycles per claimed object* are 3.6% and 1% respectively. The cause of the anomaly lies in the growth of the graph that is copied. For instance at $\mu = 6.3$, the size of the copied graph is $K = 7866$ objects, whereas it contains $K = 8080$ objects at $\mu = 7.4$. (In both of these cases $G = 1$).

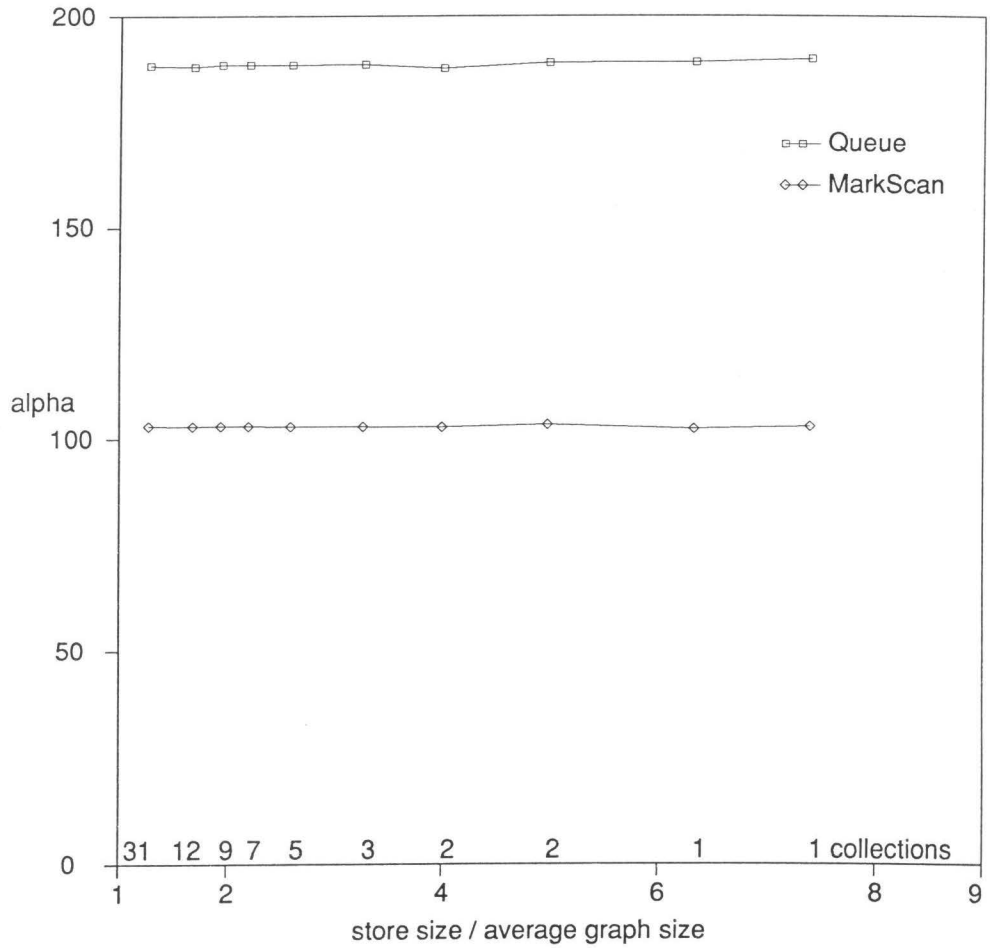


Figure 3 : Measured values of α in MC68010 processor cycles with *lambda*

With the *cycles per claimed object* we have a measure that can be used to compare the performance of garbage collection algorithms at specific values of μ. The *Queue* and *MarkScan* algorithms are better compared using the cost factors α, β and σ, because these do not depend on μ. The values of N, B, E, K and A were recorded with the *cycles per claimed object*, such that the coefficient α could be calculated from (5) using the cost factors β and σ as shown in table (2). Theoretically the values of α should be invariant to the choice of μ. However the structure and composition of the graphs is not constant during a run and the garbage collector is activated at different stages of the computation depending on the value of μ. For example the first garbage collection occurs at different instants with respect to the status of the application for the different values of μ. Subsequent garbage collections are even more uncomparable in this respect. The α values that have been computed from the

experimental data with the *lambda* application are plotted in figure (3). The horizontal axis is identical to that of figure (2).

The results with all but the smallest programs are satisfactory, since the standard deviation of α is less than 5 ‰ of the mean. The results obtained with all benchmark applications are summarised in table (3). The small programs *fib* and *quicksort* are unreliable predictors for the value of α. The reason is, that small programs exhibit a much larger variation in the percentage of shared nodes. The average values of K that were measured are presented in the row $\overline{K}$, to give an idea of the average size of the graphs in a single number.

parameter	fib	quicksort	hamming	paraff	wave	em	lambda	yacc
N	270	1035	16639	25606	389082	314520	68406	644682
B	45	134	141	1021	2113	4103	5241	14949
$\overline{K}$	62	202	593	1958	20191	14219	7650	30007
		RefCount						
cycles per object	392	365	374	394	389	382	431	412
		Queue			$\beta = 22$			
α	202	193	189	192	200	188	189	188
standard deviation	3	1	0.4	0.8	0.1	0.1	0.6	0.04
		MarkScan		$\beta = 36$	$\sigma = 50$			
α	115	113	105	105	106	98	103	100
standard deviation	1	1	0.3	0.3	0.1	0.03	0.3	0.01

Table 3 : The benchmark applications on a MC68010 using equation (5).
N, B and $\overline{K}$ in objects; α, β and σ in MC68010 processor cycles.

The α values for *Queue* give a lower bound for the case where more than a minimal object is copied (see section 4). Since most processors provide an efficient block move instruction, few more processor cycles are required to copy an extra word of information per object. With the MC68010 for each extra word (4 bytes) to be copied α must by incremented by 8.

A noticeable difference between our results and those generally accepted is the relative proximity of α and β. This may be just a matter of interpretation, since some authors are vague in this respect. Cohen writes "it is not unreasonable to assume that β is

considerably smaller than α". We find that the ratio of α to β does not exceed 10.

5.1. Other ways of calculating the cost of garbage collection

One reason to collect detailed statistics is the possibility to verify certain claims that advocates of other analysis methods make about the validity of their methods. Since our information is very fine grained it is not difficult to derive more coarse grained information from it.

We have modelled the boundary effects that are commonly ignored as the parameters E and A. We will show that if an estimate for the values of A and E is derived from the other parameters, much of the accuracy of the results is lost. The parameter E represents the number of objects that could have been claimed, but were not needed any more when the application terminates. Hence we must have that $0 \leq E < S-K$. There is no reason why E should have a particular value so we will assume its value to be average:

$$E \approx \frac{S-K}{2} = \frac{\mu-1}{2} K, G \gg 0 \tag{6}$$

The parameter A represents the total number of objects that are skipped during the scan phase of *MarkScan*. Let us assume, that when the application terminates, the remaining E free objects are distributed uniformly over the unscanned storage space. Then apart from E objects that can be claimed, there are E objects that must be skipped during the scan. With (3) we find:

$$A \approx \left(G - \frac{E}{S-K} \right) K = \frac{N+B-\mu K}{\mu-1}, G \gg 0 \tag{7}$$

Substitution of both approximations (7) and (6) in (5) yields:

$$\textit{ticks per claimed object} \approx \beta + \frac{N+B-\frac{1}{2}(\mu+1)K}{(\mu-1)N} \alpha + \frac{N+B-\mu K}{(\mu-1)N} \sigma, G \gg 0 \tag{8}$$

The first column "MC68010 approx." of table (4) has been calculated using (8). It shows a much larger standard deviation in the averages than the second column, which presents the same results but calculated using the exact equation (5) that takes the boundary effects into account. We must conclude that ignoring boundary effects gives result of significantly lower precision.

In the next four columns of table (4) the calculations with the exact equation (5) are repeated based on zero access times to various segments of the store. This provides an indication of the benefits that hardware support may bring to garbage collection. The "free stack" column gives the results that would have been obtained with an

architecture that supports a large enough high speed scratch pad to contain the entire stack needed by *RefCount* and *MarkScan*. The data indicate, that the improvement is insignificant. The fourth column applies to the situation where accessing and updating pointers in the heap is free of charge. Technically, this is a rather unrealistic assumption, but it gives some indication of the fraction of time that is spent on memory references (about 20 %). The "free code" column presents the values of α and β that can be obtained on a processor that is equipped with an instruction cache and an instruction pipeline such as the MC68020. The column "free code&stack" assumes the presence of both an instruction cache and a high speed scratch pad for the stack. Actual performance figures would not be as good, since for example a branch instruction may cause stagnation in the pipeline. We may conclude that with an instruction cache the speed of garbage collection on the MC68010 can be improved by at most a factor 5.

The last two columns of table (4) are included to verify the assumption that counting high level language statements leads to the same conclusion as precisely measuring execution time. The data in the column marked "uniform" were obtained by counting the execution of each instruction as one cycle and using the exact equation (5). An average instruction (from the subset that is exercised by the garbage collection algorithms) normally consumes about 6 processor cycles. Hence the column "uniform" gives consistently lower figures than the first two columns. In the last column we report the figures that were obtained with equation (5) by counting each high level language (Modula-2) statement as a single *tick*. Most Modula-2 statements in the implementations of the algorithms correspond to single machine instructions, making extensive use of the features of the processors instruction set. Some conditionals require two instructions: one to set the condition codes and the other to branch on the appropriate condition.

The last row of table (4) allows for the various counting methods to be compared. The performance of the *MarkScan* and *Queue* algorithms is best compared using the values of α, because α represents the work performed by the traversal algorithm in the garbage collector. We have chosen to use the *MarkScan* algorithm as a reference point because it has the best performance. From the values obtained for the α-ratio we find that all counting methods that we have used lead to the same result $\alpha_{Queue} / \alpha_{MarkScan} = 1.76 \pm 8\%$.

It should be noted, that the *cycles per claimed object* ratio of the two algorithms is generally less than 1.76, because $\beta_{MarkScan} > \beta_{Queue}$ and the *MarkScan* algorithm also suffers from the cost of skipping marked objects. At high μ values the programs from our benchmark show a slight advantage for *MarkScan* (see for example figure 2).

	eq (8)	eq (5)						
parameter	MC68010 approx.	MC68010 exact	free stack	free heap	free code&stack	free code	uniform	Modula-2
RefCount								
cycles per object	455	455	455	357	98	98	81.5	65.3
Queue								
α	193	189	189	155	33.4	33	30	24.3
standard deviation	28	0.6	0.6	0.6	0.1	0.1	0.2	0.1
β	22	22	22	22	0	0	2	2
MarkScan								
α	113	103	101	85	19.5	18	18.5	13.8
standard deviation	24	0.3	0.2	0.2	0.1	0.1	0.1	0.1
β	36	36	36	32	4	4	5	4
σ	50	50	50	42	8	8	7	5
$\frac{\alpha_{Queue}}{\alpha_{MarkScan}}$	1.70	1.83	1.87	1.82	1.71	1.83	1.62	1.76

Table 4 : The *lambda* program with various measurement methods
α, β and σ in MC68010 processor cycles (except the last two columns).

6. Conclusions

Experiments with garbage collection algorithms have been conducted in an unconventional way. The mutator is a fixed combinator graph reducer which is driven by a benchmark of SASL programs. We have thus used an average application to drive a worst case list processing system. As a consequence our selection of three garbage collection algorithms had to cope with the heaviest possible demand of free objects while the structure of the graphs that they had to process was real.

Three garbage collection methods were implemented in an optimal way on a Motorola MC68010 processor. We have measured the performance of reference counting, mark/scan and copying garbage collection at subinstruction level. The performance of reference counting garbage collection was found to be superior to that of the other algorithms if the graph being manipulated fits snugly into the store ($\mu \approx 1$). The performance of all algorithms is roughly equal when the store provides about twice as

much space as the average graph requires. A simple explanation for this effect is, that reference counting algorithms visit mostly garbage, whereas the copying and to a lesser extent mark/scan algorithms visit objects that are still in use. Hence the costs should be roughly the same, when the the amount of garbage is in equilibrium with the number of reachable objects. The mark/scan algorithm is a little faster than the two-space copy algorithm.

The conclusions that were drawn based on these measurements can also be reached if higher level events are counted. One way to achieve this is to count machine instructions and memory references. Counting high level language statements requires more care, since statements of an arbitrary complexity are easily constructed. One must write the high level language code with the target processor in mind.

In our performance measurements we have taken all boundary effects into account and show that adding more store to a list processing system with a non-reference counting garbage collector does not necessarily lower the cost of storage allocation and garbage collection. The explanation for form of anomalous behaviour is, that adding more store generally postpones the moment at which the garbage collector is activated. Since the size and composition of the graph to be traversed is not constant, it entirely depends on the state of the graph how much time it will cost to traverse the graph.

In the experiments we have imposed two restrictions on the graphs that require garbage collection: reclamation of cyclic structures is unnecessary and all objects are of the same size. Our conclusions remain valid if we alleviate the first restriction. The non-reference counting algorithms always properly deal with cycles, hence the costs involved can not change. The cost of reference counting, which is already the most expensive garbage collection method, will yet increase. If the second restriction is dropped the cost of mark/scan and reference count will increase more than that of the two-space copy method, because the latter requires much less modification to cope with varisized cells.

Acknowledgements

Discussions with Arthur Veen, Wim Vree and Betsy Pepels have provided the stimulus to explore this area of research. Betsy made the suggestion to include Cheney's algorithm. Arthur and Wim made valuable comments on draft versions of the paper.

References

1. J. Cohen, "Garbage collection of linked structures," *Computing Surveys* **13**(3) pp. 341-367 (Sep. 1981).

2. D. P. Friedman and D. S. Wise, ''Reference counting can manage the circular environments of mutual recursion,'' *Information processing letters* **8**(1) pp. 41-45 (Jan. 1979).

3. D. R. Brownbridge, ''Cyclic reference counting for combinator machines,'' pp. 273-288 in *Second conf. on functional programming languages and computer architecture, LNCS 201*, ed. J.-P. Jouannaud,Springer verlag, Nancy, France (Sep. 1985).

4. D. E. Knuth, *The art of computer programming, volume 1: Fundamental algorithms*, Addison Wesley, Reading, Massachusetts (1973). second edition

5. J. L. Baer and M. Fries, ''On the efficiency of some list marking algorithms,'' pp. 751-756 in *Information processing 77*, ed. B. Gilchrist,North Holland, Toronto, Canada (Aug. 1977).

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

7. P. H. Hartel and A. H. Veen, ''Statistics on graph reduction of SASL programs,'' *Software practice and experience* **18**(3) pp. 239-253 (Mar. 1988).

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

9. D. A. Turner, ''The semantic elegance of applicative languages,'' pp. 85-92 in *Conf. on Functional programming languages and computer architecture*, ed. Arvind,ACM, Portsmouth, New Hampshire (Oct. 1981).

10. W. G. Vree, ''The grain size of parallel computations in a functional program,'' pp. 363-370 in *Conf. on Parallel processing and Applications*, ed. E. Chiricozzi, A. d'Amico,Elsevier Science Publishing, L'Aquila, Italy (Sep. 1987).

11. P. W. M. Koopman, ''Interactive programs in a functional language: A functional implementation of an editor,'' *Software practice and experience* **17**(9) pp. 609-622 (Sep. 1987).

12. H. Glaser, C. Hankin, and D. Till, *Principles of functional programming*, Prentice Hall, Englewood Cliffs, New Jersey (1984).

13. S. L. Peyton Jones, ''Yacc in SASL - an exercise in functional programming,'' *Software practice and experience* **15**(8) pp. 807-820 (Aug. 1985).

14. D. A. Turner, ''A new implementation technique for applicative languages,'' *Software Practice and Experience* **9**(1) pp. 31-49 (Jan. 1979).

15. T. Johnsson, ''Efficient compilation of lazy evaluation,'' *Sigplan Notices* **19**(6) pp. 58-69 (Jun. 1984).

16. 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 *Third conf. on functional programming languages and computer architecture, LNCS 274*, ed. G. Kahn,Springer verlag, Portland, Oregon (Sep. 1987).

17. J. Fairbairn and S. Wray, "Tim: A simple lazy abstract machine to execute super-combinators," pp. 34-45 in *Third conf. on functional programming languages and computer architecture, LNCS 274*, ed. G. Kahn,Springer verlag, Portland, Oregon (Sep. 1987).

18. P. H. Hartel, *Performance analysis of storage management in combinator graph reduction,* Dept. of Comp. Sys, Univ. of Amsterdam (Oct. 1988). PhD. Thesis

19. W. G. Vree, "Parallel graph reduction for communicating sequential processes," PRM project internal report D-26, Dept. of Comp. Sys, Univ. of Amsterdam (Aug. 1988).

20. C. J. Cheney, "A non-recursive list compacting algorithm," *CACM* **13**(11) pp. 677-678 (Nov. 1970).

21. C. A. R. Hoare, "Optimization of store use for garbage collection," *Information processing letters* **2**(1) pp. 165-166 (1974).

22. Motorola Inc., *MC68000 8-/16-/32-Bit microprocessors programmer's reference manual,* Prentice hall, Englewood Cliffs, New Jersey (1986). fifth edition

The average size of ordered binary subgraphs

Pieter H. Hartel

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

Abstract

To analyse the demands made on the garbage collector in a graph reduction system, the change in size of an average graph is studied when an arbitrary edge is removed. In ordered binary trees the average number of deleted nodes as a result of cutting a single edge is equal to the average size of a subtree. Under the assumption that all trees with n nodes are equally likely to occur, the expected size of a subtree is found to be approximately $\sqrt{\pi n}$. The enumeration procedure can be applied to graphs by considering spanning trees in which the nodes that were shared in the graph are marked in the spanning tree. A correction to the calculation of the average is applied by ignoring subgraphs that have a marked root. Under the same assumption as above the average size of a subgraph is approximately $\sqrt{\pi n} - 2(m+1)$, where m represents the number of shared nodes and $m \ll n$.

Key words: binary graphs Catalan statistics combinator graph reduction subgraphs

1. Introduction

The λ-calculus[1] can be viewed as a universal programming language. Its simplicity makes λ-expressions (programs) amenable to direct mechanical evaluation.[2] Functional programming languages in essence are "sugared" versions of the λ-calculus. An ordered binary tree provides a natural representation for a λ-expression. A function application (a node) consists of the juxtaposition of the function (the left descendant of the node) and its argument (the right descendant of the node). A subexpression appears as a subtree, hence arbitrarily complex expressions can be represented. Graph reduction[3] is generally preferred to tree reduction because it allows subexpressions to be shared in stead of copied. This saves both space and time, since a copy of an expression can be made by creating a new pointer to the representation of the expression. This improved efficiency is not without cost.

Let us assume, that initially a λ-expression is presented for mechanical evaluation, which contains at least one reducible expression (redex). Furthermore let there be a mechanism that decides which redex is to be evaluated next. The evaluation process then consists of a number of discrete reduction steps (e.g. β-reduction, α-conversion),

which take the expression to its final form, or at least a number of steps ahead. During this process the expression may take many different forms, which although semantically equivalent, require a varying amount of space for their graphical representation. Each reduction step typically causes some nodes to be deleted and some to be added to the graph. New nodes are added explicitly, for example as a result of β-reduction, but old nodes become unreachable without explicit notice. The reason is, that unless special precautions are taken, we can not know when the last reference to a possibly shared node is destroyed. This uncertainty has an important consequence for practical implementations of the λ-calculus and λ-based languages, because it necessitates a garbage collector, i.e. a device that recuperates storage occupied by parts of the graph that have become unreachable from the root. The cost of garbage collection is considerable. It may take up to an order of magnitude more time to recuperate the storage occupied by a node than it takes to allocate a node from a pool of free nodes.[4] Many methods have been proposed to control the cost of garbage collection.[5]

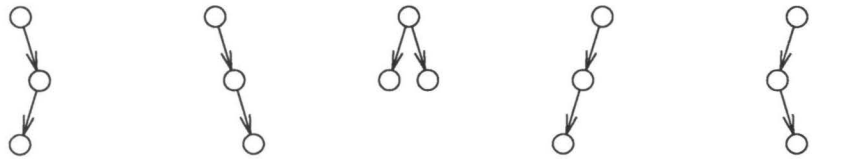
To arrive at a better understanding of the cost of garbage collection we will count the number of nodes that may be expected to turn into garbage during a reduction step. If we are prepared to make some assumptions about the structure of the graphs being manipulated, interesting properties can be derived, even if the precise configuration is not known. For instance, under the assumption that all ordered trees with n nodes are equally likely to occur, it can be shown[6] that the average height of such trees is approximately $\sqrt{\pi n}$. The pre-order spanning trees of the graphs that occur during the evaluation of complicated expressions were found to behave in roughly the same way: their average height is proportional to $\sqrt{\pi n}$.[7] Both ways of arriving at an average apparently lead to comparable results.

A method is presented that allows us to extend the results about ordered binary trees (Catalan statistics) to the graphs that occur during graph reduction. As a first approximation we regard graphs as trees, i.e. ignore the effects of sharing completely. This allows us to apply the basic Catalan statistics directly to our graphs (next section). In section 3 the effect of sharing is modelled by marking the nodes in a spanning tree that correspond to shared nodes in the program graph. New counting procedures are developed to take sharing into account. The results give a lower bound and an upper bound on the average number of nodes in a subgraph.

2. Enumeration of ordered binary trees

In the rest of the paper we assume, that the reader is familiar with the analysis as presented by Knuth [8] pp. 388 - 389. To summarise the most important results, the construction method for ordered binary trees and the formulae for b_n and $B(z)$ are reproduced here. An ordered binary tree with n nodes can be built by taking a root and

attaching an ordered binary tree with i nodes to the left and an ordered binary tree with $n-i-1$ nodes to the right. To construct the set of ordered binary trees with n nodes, i must be varied over the range $[0..n-1]$ and the construction must be applied recursively to the subtrees. As an example, consider the set of ordered binary trees with 3 nodes shown in figure (1).



(a) $i=0$ (b) $i=0$ (c) $i=1$ (d) $i=2$ (e) $i=2$

Figure 1 : The set of ordered binary trees with 3 nodes

The recurrence relation and the boundary conditions for $b_n \equiv$ the number of ordered binary trees with n nodes are:

$$b_n = \left\{ \begin{array}{ll} b_0 b_{n-1} + b_1 b_{n-2} + \cdots + b_{n-2} b_1 + b_{n-1} b_0, & n > 0 \\ 1, & n = 0 \end{array} \right\} \tag{1}$$

The generating function $B(z)$ for the sequence $\langle b_n \rangle$ is:

$$B(z) = \sum_{n=0}^{\infty} b_n z^n = \frac{1-\sqrt{1-4z}}{2z}$$

The closed form for the number of ordered binary trees with n nodes:

$$b_n = \frac{1}{n+1} \left[\begin{matrix} 2n \\ n \end{matrix} \right] \tag{2}$$

With Stirling's approximation to $n!$ for large n we have:

$$b_n = \frac{4^n}{n\sqrt{\pi n}} + O\left(\frac{4^n}{n^2 \sqrt{n}} \right)$$

This concludes the exposé of the basic results in Catalan statistics.

2.1. The size of ordered binary subtrees

The collection of nodes that become unreachable from the root when an edge is cut is called a subtree. In addition the entire tree is considered as a subtree. As a consequence there is one subtree associated with each node in a tree. Let $s_{n,k}$ be the number of subtrees with k nodes of the set of ordered binary trees with n nodes. Suppose we

want to find the number of subtrees with 2 nodes in the set of trees with 3 nodes. Counting subtrees in figure (1) yields the answer $s_{3,2} = 4$. The configurations (a), (b), (d) and (e) each contribute one to the total. In general it is not practical to draw all configurations in order to count subtrees, hence we must use the construction method to arrive at the desired answer. The configurations (a) and (b) were constructed by attaching all possible configurations with two nodes to the right of the root combined with all possible configurations with zero nodes to the left. Together they contribute $b_0 s_{2,2} + s_{0,2} b_2$ to the number of subtrees with two nodes. Extending this procedure to the remaining configurations yields:

$$s_{3,2} = b_0 s_{2,2} + b_1 s_{1,2} + b_2 s_{0,2} + s_{0,2} b_2 + s_{1,2} b_1 + s_{2,2} b_0$$

$$= 1\ s_{2,2} + 1\ s_{1,2} + 2\ s_{0,2} + s_{0,2}\, 2\ + s_{1,2}\, 1\ + s_{2,2}\, 1\ = 4$$

The general rule is derived from the enumeration formula for ordered binary trees: where b_i is multiplied by b_{n-i-1} in (1), we must now multiply $s_{i,k}$ by b_{n-i-1}. This corresponds to the number of subtrees with k nodes in the set of trees with i nodes, multiplied by the number of times this configuration occurs to the left of the root. Similarly enumeration of subtrees with k nodes in the right subtree yields the term $b_i\ s_{n-i-1,k}$. Since a tree is regarded as a subtree we have $s_{n,n} = b_n$. The recurrence relation and the boundary conditions for $s_{n,k} \equiv$ the number of subtrees with k nodes among the set of trees with n nodes are:

$$s_{n,k} = \left\{ \begin{array}{ll} b_0 s_{n-1,k} + b_1 s_{n-2,k} + \cdots + b_{n-2} s_{1,k} + b_{n-1} s_{0,k} + & \\ s_{0,k} b_{n-1} + s_{1,k} b_{n-2} + \cdots + s_{n-2,k} b_1 + s_{n-1,k} b_0\,, & 0 < k < n \\ b_n\,, & n \geq 0 \wedge k = n \\ 0\,, & \text{otherwise} \end{array} \right\} \qquad (3)$$

To derive the closed form of (3), two auxiliary results are needed. Both can be proved by induction on n from (3). The first yields the number of subtrees with one node:

$$s_{n,1} = n\ b_{n-1} \qquad (4)$$

Proof: if n=1 we have $s_{1,1} = b_1 = 1\ b_0$, since $b_0 = b_1 = 1$. Application of the induction hypothesis $\forall i \in 1..n-1 : s_{i,1} = i\ b_{i-1}$ to (3) yields

$$
\begin{aligned}
s_{n,1} = \quad & b_0\, s_{n-1,1} \quad + \cdots + b_{n-2}\, s_{1,1} \quad + b_{n-1}\, s_{0,1} + \\
& s_{0,1}\, b_{n-1} + s_{1,1}\, b_{n-2} \quad + \cdots + s_{n-1,1}\, b_0 \\
= \quad & b_0\,(n-1)\, b_{n-2} + \cdots + b_{n-2}\, 1\, b_0 \quad + b_{n-1}\, 0 \quad + \\
& 0\, b_{n-1} \quad + 1\, b_0\, b_{n-2} \quad + \cdots + (n-1)\, b_{n-2}\, b_0
\end{aligned}
$$

$$= n\ (b_0\, b_{n-2} + b_1\, b_{n-3} + \cdots + b_{n-2}\, b_0)$$

$$= n\ b_{n-1}$$

The second auxiliary result (5) can be proved in a similar fashion.

$$s_{n,k} = s_{n-k+1,1}\ s_{k,k}\ , \quad 0 < k \le n \tag{5}$$

Combining (3), (4) and (5) we find:

$$s_{n,k} = (n-k+1)\, b_{n-k}\, b_k\ , \quad 0 < k \le n \tag{6}$$

The generating function for the sequence $\langle\, s_{n,k}\,\rangle$ with a fixed value of k is:

$$S_k(z) = \sum_{n=0}^{\infty} s_{n,k} z^n = \sum_{n=0}^{\infty} (n-k+1)\, b_{n-k}\, b_k z^n = b_k z^k\, \frac{d}{dz}(z\, B(z)) = \frac{b_k z^k}{\sqrt{1-4z}}\ , \quad k > 0$$

Since every subtree is uniquely determined by its top node, the total number of subtrees must be equal to the total number of nodes in the set of b_n trees. This can be proved directly from (3).

$$\sum_{k=1}^{n} s_{n,k} = n\ b_n$$

The values of $s_{n,k}$ grow quickly as n increases. Table (1) shows the values for small n and k. The numbers on the diagonal are the values for b_n (Catalan numbers) since $s_{n,n} = b_n$.

n	k=1	k=2	k=3	k=4	k=5	k=6	k=7	k=8	k=9
1	1								
2	2	2							
3	6	4	5						
4	20	12	10	14					
5	70	40	30	28	42				
6	252	140	100	84	84	132			
7	924	504	350	280	252	264	429		
8	3432	1848	1260	980	840	792	858	1430	
9	12870	6864	4620	3528	2940	2640	2574	2860	4862

Table 1: $s_{n,k}$ for small values of n and k.

2.2. Average size of subtrees

Under the assumption that all ordered binary trees with n nodes occur with the same probability, we find for the size of the average subtree:

$$\overline{s_n} = \frac{\sum_{k=0}^{n} k\, s_{n,k}}{n\, b_n} = \frac{\sum_{k=0}^{n} k\,(n-k+1)\, b_{n-k}\, b_k}{n\, b_n} \tag{7}$$

The generating function for the sequence $\langle \sum_{k=0}^{n} k\, s_{n,k} \rangle$ is:

$$\sum_{n=0}^{\infty} \sum_{k=0}^{n} (n-k+1)\, b_{n-k}\, k\, b_k z^n = z\, B'(z)\, \frac{d}{dz}(z\, B(z))$$

$$= \frac{1}{1-4z} + \frac{1}{2z} - \frac{1}{2z\sqrt{1-4z}}$$

$$= \sum_{n=0}^{\infty} 4^n z^n + \frac{1}{2z} - \frac{1}{2z} \sum_{n=0}^{\infty} \binom{-1/2}{n} (-4)^n z^n$$

$$= \sum_{n=0}^{\infty} \left[4^n - \frac{1}{2} \binom{-1/2}{n+1} (-4)^{n+1} \right] z^n$$

$$= \sum_{n=0}^{\infty} \left[4^n - (2n+1)\, b_n \right] z^n$$

With this result (7) becomes:

$$\overline{s_n} = \frac{4^n - (2n+1)\, b_n}{n\, b_n}, \quad n > 0$$

Let us apply this result to graph reduction. Using Stirling's approximation we find for large n that $\overline{s_n} \simeq \sqrt{\pi\, n}$. Hence a reduction step may be expected to delete a subgraph containing $\sqrt{\pi\, n}$ nodes from the program graph. Unless a reduction step adds a similar number of new nodes to the graph, it is reduced in size until the average number of nodes added per reduction step and that deleted are in equilibrium. From our experiments we know that such an equilibrium may be reached when far fewer than $\sqrt{\pi\, n}$ nodes are added per reduction step. The effect of sharing that is completely ignored in the current approximation, should at least be taken into account.

3. Trees with marked nodes

Our objective is to count the number of nodes that may be deleted from a program graph by cutting a single edge. To be able to apply the results obtained so far, we must eliminate the effect of the extra edges that turn a tree into a graph. In general there are many subsets of edges that may be considered redundant. From our experience with the determination of the average height of spanning trees we found that the results are virtually independent of the particular set of edges that is removed. We have no reason to believe that in the current case the results should be sensitive to the choice of spanning tree, since the averaging processes in determining the expected height of trees and the expected size of subtrees are so similar. The first step is therefore to consider an arbitrary spanning tree of a graph in stead of the graph itself. Such a tree has the same set of nodes as the graph and one fewer edge than the graph has nodes. In the previous section we saw that without further refinement the results with binary subtrees are still unsatisfactory. We therefore introduce a "marking" mechanism by which we can remember which nodes were originally shared. The enumeration procedures must take this marking into account.

There are two issues that deserve further attention. In the first place we have to cope with nodes that have a different number of incident edges. In the second place the

origin of the edges incident upon the same node may be expected to play a role. Since we wish to study the effect of removing a single edge from a graph, we need not be too concerned about the actual number of incident edges to a node as long as it is greater than one. Usually a shared node remains reachable from the root if an incident edge is removed. A connected component in a graph may be attached to the rest of the graph via a single edge. If that edge is cut, the entire component becomes unreachable from the root. However, since our experiments indicate that cycles are rare in practical graph reduction, we ignore this effect and treat a shared node as one that can not be removed by cutting a single edge.

(a) removable marked node (b) permanent marked node

Figure 2 : Two binary graphs with one marked node

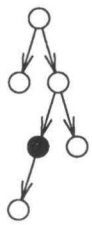
The origin of the edges can not be completely ignored. Figure (2-a) shows, that a marked node may originate from two different kinds of graphical structure. If the left descendant of the root (marked with the letter "x") is removed, the right subgraph of node "x" also becomes disconnected, since all edges incident upon the shared node are successively removed. The right subgraph of node "x" remains connected if it is removed from the graph shown in figure (2-b). The marked nodes in trees corresponding to the graph as in figure (2-a) are called removable, the type of marking as in figure (2-b) is called permanent. We will study both kinds of marking.

Based on the graphical configurations as illustrated in figure (2) we develop the rules by which subtrees of marked trees are enumerated. Ultimately we are interested in the expected size of subtrees if all trees are equally likely to occur. The averaging process therefore considers every node once as the root of a subtree that must be counted:

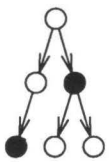
- If the root of a subtree is marked the subtree is not counted, regardless of the type of marking. This is in accordance with the origin of a marked node as a shared node in a program graph.
- If the root of a subtree is not marked, the subtree is counted, *including* the removable marked nodes and their subtrees but *excluding* the permanent marked nodes and their subtrees.

Consider as an example the trees of figure (3). If the marking in figure (3-a) is considered permanent, there are 3 subtrees with one node, 1 subtree with two nodes and 1

subtree with four nodes. If the marking is considered removable, there are 3 subtrees with one node, 1 with four and 1 with six nodes. Similarly with permanent marking figure (3-b) shows 3 subtrees of one and 1 subtree of two nodes. With removable marking we count 2 subtrees of one, 1 of two and 1 of six nodes.



(a) one marked node



(b) two marked nodes

Figure 3 : Two marked binary trees

A more careful consideration of the graphical structure in figure (2) reveals that the permanent or removable status of a marked node is a property that varies with the choice of the first node to be deleted. For example if rather than the node "x" we remove the root in figure (2-b), even the permanent node and its subtree should disappear. However we can afford to ignore this complication, since the expected size of subtrees with only permanent or only removable marked nodes form the extremes of a range that encompasses realistic values of the expected number of nodes to be deleted in program graphs (provided the other assumptions prove to be realistic).

It will be shown, that in a tree with only removable marked nodes, the enumeration for unmarked trees can be modified in a straight forward manner by discounting all subtrees with a marked root. Permanent marking causes considerable complication since each subtree has to be "searched" for permanent marked nodes, which "truncate" the subtree to a smaller size. First the results are generalised to reason about trees with one removable or permanent marked node. This allows us to develop the necessary intuition to solve the problem if an arbitrary number of nodes is marked.

3.1. The size of subtrees in trees with one removable marked node

In the collection of the $n\ b_n$ trees with one removable marked node, each subtree appears once with a marked root and $n-1$ times with an unmarked root. Since the total number of subtrees with k nodes in the $n\ b_n$ trees with n nodes is $n\ s_{n,k}$, the number of subtrees without marked nodes is:

$$r_{n,k} = (n-1)\, s_{n,k}\ , \quad n > 0$$

The total number of subtrees must be equal to the total number of nodes in all $n\ b_n$

trees, minus the number of marked nodes:

$$\sum_{k=1}^{n} r_{n,k} = n\,(n-1)\,b_n$$

3.2. Average size of subtrees in trees with one removable marked node

Making the same assumption as before about the probability distribution, we find that the average size of subtrees in trees with one removable marked node is:

$$\overline{r_n} = \frac{\sum_{k=0}^{n} k\, r_{n,k}}{n^2\, b_n} = \frac{(n-1)\sum_{k=0}^{n} k\, s_{n,k}}{n^2\, b_n} = \left(1-\frac{1}{n}\right)\overline{s_n}, \qquad n > 0 \tag{8}$$

3.3. The size of subtrees in trees with one permanent marked node

The construction method for trees is extended to take the location of the marked node into account. Any one of the n nodes in a tree may be marked. Therefore, the total number of trees that must be considered is $n\, b_n$. The recurrence relation of (1) is rewritten to reflect this situation:

$$n\, b_n = \left[b_0\,(n-1)\,b_{n-1} + 0\, b_0\, b_{n-1}\right] + \left[b_1\,(n-2)\,b_{n-2} + 1\, b_1\, b_{n-2}\right] + \cdots$$

$$+ \left[b_{n-1}\, 0\, b_0 + (n-1)\, b_{n-1}\, b_0\right] + b_n \tag{9}$$

In the pairs of terms, the first subterm corresponds to the configuration with the marked node residing in the right subtree. The second subterm has the same relation to the left subtree. Both subterms therefore represent configurations with an unmarked root. The last term of (9) enumerates the b_n trees of n nodes with a marked root.

The recurrence relation and boundary conditions for $p_{n,k} \equiv$ the number of subtrees with k nodes in the set of trees with n nodes of which one is permanent marked can now be formulated by enumerating the relevant number of subtrees in each term of (9):

$$p_{n,k} = \left\{ \begin{array}{ll} 2\,(\ b_0 p_{n-1,k} + \ b_1 p_{n-2,k} + \cdots + \ b_{n-1} p_{0,k}) + \\ 2\,(0\,b_0 s_{n-1,k} + 1\,b_1 s_{n-2,k} + \cdots + (n-1)\,b_{n-1} s_{0,k}) + \\ s_{n,k} + s_{n,n-k}\ , & 0 < k < n \\ 1\ , & n = k = 0 \\ 0\ , & \text{otherwise} \end{array} \right\} \qquad (10)$$

The first two subterms of (10) correspond to configurations with an unmarked root. The marked node either resides in the left subtree or in the right subtree. The second subterm accounts for the fact that if one subtree (say with size i) contains the marked node, the other (with size $n-i-1$) is unmarked. This gives a contribution of $i\ b_i\ s_{n-i-1,k}$ subtrees, taken over all possible values of i. The factor i originates from the fact that the marked node may reside at any one of the i places in the subtree.

If the root of the tree is marked, the situation resembles that of the unmarked trees. This accounts for the subterm $s_{n,k}$, but with $k < n$, since there are no subtrees with n nodes in any marked tree with n nodes. The last subterm of (10) takes into account, that each subtree with a marked root of size k "prunes" a branch of the main tree, such that an unmarked tree with $n-k$ nodes remains.

From (10) we find the open form of the generating function for the sequence $\langle\, p_{n,k}\,\rangle$ with a fixed value of k:

$$P_k(z) = \sum_{n=0}^{\infty} \left[2 \sum_{i=0}^{n-1} b_i\, p_{n-i-1,k} + 2 \sum_{i=0}^{n-1} i\, b_i\, s_{n-i-1,k} + s_{n,k} + s_{n,n-k} \right] z^n$$

$$- \left[s_{k,k} + s_{k,0} \right] z^k \qquad (11)$$

The correction term in (11) is necessary to compensate for $p_{k,k}$, which if the recurrence relation in (10) is used for $k=n$ (hence outside its domain) yields $s_{k,k}+s_{k,0}$ instead of 0.

From (6) we find that:

$$s_{n,k} + s_{n,n-k} = (n-k+1)\, b_{n-k}\, b_k + (k+1)\, b_k\, b_{n-k} = (n+2)\, b_k\, b_{n-k}$$

Substitution of this result in (11) yields:

$$P_k(z) = \sum_{n=0}^{\infty} \left[2 \sum_{i=0}^{n-1} b_i \, p_{n-i-1,k} + 2 \sum_{i=0}^{n-1} i \, b_i \, s_{n-i-1,k} + (n+2) \, b_k \, b_{n-k} \right] z^n - (k+2) \, b_k \, b_0 \, z^k$$

$$= 2 \, z \, B(z) \, P_k(z) + 2 \, z^2 \, B'(z) \, S_k(z) + \frac{b_k}{z} \frac{d}{dz} \left[z^{k+2} \, (B(z) - 1) \right]$$

This equation can be solved for $P_k(z)$ yielding:

$$P_k(z) = \frac{1}{1 - 2z \, B(z)} \left[2 \, z^2 \, B'(z) \, S_k(z) + \frac{b_k}{z} \frac{d}{dz} \left[z^{k+2} \, (B(z) - 1) \right] \right]$$

$$= b_k \, z^k \left[\frac{k+1}{2z} \left[\frac{1}{\sqrt{1-4z}} - 1 \right] - \frac{k+1}{\sqrt{1-4z}} + \frac{2z}{(1-4z)^{1½}} \right]$$

$$= b_k \, z^k \left[\frac{k+1}{2} \sum_{n=0}^{\infty} \binom{-½}{n+1} (-4)^{n+1} z^n - (k+1) \sum_{n=0}^{\infty} \binom{-½}{n} (-4z)^n \right.$$

$$\left. + 2 \sum_{n=1}^{\infty} \binom{-1½}{n-1} (-4)^{n-1} z^n \right]$$

$$= \sum_{n=0}^{\infty} (n+2) \, (n-k) \, b_{n-k} \, b_k \, z^n$$

Hence:

$$p_{n,k} = (n+2) \, (n-k) \, b_{n-k} \, b_k \, , \quad 0 < k \leq n \tag{12}$$

The total number of subtrees must be equal to the total number of nodes in all $n \, b_n$ trees minus the number of marked nodes:

$$\sum_{k=1}^{n} p_{n,k} = (n-1) \, n \, b_n$$

3.4. Average size of subtrees in trees with one permanent marked node

With a uniform probability distribution, the average size of subtrees in trees with one permanent marked node is:

$$\overline{p_n} = \frac{\sum_{k=0}^{n} k\, p_{n,k}}{n^2\, b_n} = \frac{\sum_{k=0}^{n} k\,(n+2)\,(n-k)\, b_{n-k}\, b_k}{n^2\, b_n} \tag{13}$$

The generating function for the sequence $\langle \sum_{k=0}^{n} k\, p_{n,k} \rangle$ is:

$$\sum_{n=0}^{\infty} \sum_{k=0}^{n} (n+2)\,(n-k)\, b_{n-k}\, k\, b_k\, z^n = \frac{1}{z} \frac{d}{dz}(z^2 B'(z))^2$$

$$= \frac{2\,(1-2z)}{(1-4z)^2} - \frac{2}{(1-4z)^{1½}} \tag{14}$$

$$= \sum_{n=1}^{\infty} \left[(n+2)\, 4^n - 2 \begin{bmatrix} -1½ \\ n \end{bmatrix} (-4)^n \right] z^n$$

After simplification of the binomial coefficient, the result can be substituted in (13) such that:

$$\overline{p_n} = \frac{(n+2)\, 4^n - (2n+1)\,(2n+2)\, b_n}{n^2\, b_n}, \quad n > 0$$

3.5. The size of removable marked subtrees

In the set of $\begin{bmatrix} n \\ m \end{bmatrix} b_n$ trees with m removable marked nodes, each subtree appears $\begin{bmatrix} n-1 \\ m-1 \end{bmatrix}$ times with a marked root. The total number of subtrees with k nodes in the $\begin{bmatrix} n \\ m \end{bmatrix} b_n$ trees is $\begin{bmatrix} n \\ m \end{bmatrix} s_{n,k}$. Therefore $r^m_{n,k} \equiv$ the number of subtrees with k nodes in the set of trees with n nodes, of which m are removable marked is:

$$r^m_{n,k} = \left(\begin{bmatrix} n \\ m \end{bmatrix} - \begin{bmatrix} n-1 \\ m-1 \end{bmatrix} \right) s_{n,k} = \begin{bmatrix} n-1 \\ m \end{bmatrix} s_{n,k}, \quad n > 0$$

The total number of subtrees must be equal to the total number of nodes in the set of $\begin{bmatrix} n \\ m \end{bmatrix} b_n$ trees, minus the number of marked nodes:

$$\sum_{k=1}^{n} r_{n,k}^{m} = \begin{bmatrix} n \\ m \end{bmatrix} (n-m)\, b_n \tag{15}$$

3.6. Average size of removable marked subtrees

With a uniform probability distribution, the average size of subtrees in removable marked trees is:

$$\overline{r_n^m} = \frac{\sum_{k=0}^{n} k\, r_{n,k}^{m}}{\begin{bmatrix} n \\ m \end{bmatrix} b_n} = \frac{\begin{bmatrix} n-1 \\ m \end{bmatrix} \sum_{k=0}^{n} k\, s_{n,k}}{\begin{bmatrix} n \\ m \end{bmatrix} n\, b_n} = \left(1-\frac{m}{n}\right) \overline{s_n}, \qquad n > 0$$

3.7. The size of permanent marked subtrees

The generalisation of the result to trees with an arbitrary number of permanent marked nodes follows along the lines set out in the previous paragraphs. We commence by characterising the subtrees of all $\begin{bmatrix} n \\ m \end{bmatrix} b_n$ marked trees, by the way the m permanent marked nodes are distributed over the total n nodes. Using Vandermonde's convolution[8] and (1) we find that:

$$\begin{bmatrix} n \\ m \end{bmatrix} b_n = \begin{bmatrix} n-1 \\ m \end{bmatrix} b_n + \begin{bmatrix} n-1 \\ m-1 \end{bmatrix} b_n$$

$$= \sum_{i=0}^{m} \sum_{j=0}^{n-1} \begin{bmatrix} j \\ i \end{bmatrix} b_j \begin{bmatrix} n-j-1 \\ m-i \end{bmatrix} b_{n-j-1} + \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} \begin{bmatrix} j \\ i \end{bmatrix} b_j \begin{bmatrix} n-j-1 \\ m-i-1 \end{bmatrix} b_{n-j-1} \tag{16}$$

The first sum in (16) corresponds to configurations with an unmarked root, the second to those with a marked root. In both "inner" sums, the index j ranges over all possible combinations of subtrees. The "outer" sums distribute the marked nodes over the subtrees, starting with all marked nodes to the left and none to the right, one to the left and all but one to the right etc.

Looking at figure (3) again and interpreting the marked nodes as permanent, it appears that there are two kinds of subtrees eligible to be counted. If the root of the main tree is unmarked, it is counted as a subtree. This subtree is called a top-tree. It will prove useful, to extend the definition of top-trees to the case where the root of main tree *is* marked. In that case a top-tree is considered to have 0 nodes. The remaining subtrees (i.e. non top-trees) are isolated from the root by a marked node. In figure (3-a) there is one subtree of either kind and in (3-b) there are two isolated subtrees (if subtrees of

top-trees are ignored). The recurrence relation and the boundary conditions for $t_{n,k}^{m} \equiv$ the number of top-trees with k nodes in the set of trees with n nodes, of which m nodes are permanent marked are:

$$t_{n,k}^{m} = \begin{cases} \sum_{h=0}^{k-1} \sum_{j=0}^{n-1} \sum_{i=0}^{m} t_{j,h}^{i}\, t_{n-j-1,k-h-1}^{m-i}, & n>0 \wedge m>0 \wedge k>0 \wedge k+m \le n \\ \binom{n-1}{m-1} b_n, & n>0 \wedge m>0 \wedge k=0 \\ b_n, & n \ge 0 \wedge m=0 \wedge k=n \\ 0, & \text{otherwise} \end{cases} \qquad (17)$$

The recurrence in the above expression draws upon the "unmarked root" sum in (16), with the constraint that if the left subtree has h nodes, the right subtree must host the remaining $k-h-1$ nodes. The second clause in (17) gives the number of times that the root is marked. This is the number of times a top-tree with 0 nodes must be counted in a marked tree.

We conjecture that from (17) it can be proved by induction:

$$t_{n,k}^{m} = \frac{(k+1)}{n-k} \binom{n+m}{m-1} \binom{2n-2k}{n-k-m} b_k, \quad n>0 \wedge m>0 \wedge k \ge 0 \wedge k+m \le n \qquad (18)$$

With this result we can formulate the recurrence relation and the boundary conditions for $p_{n,k}^{m} \equiv$ the number of subtrees with k nodes in the set of trees with n nodes, of which m nodes are permanent marked:

$$p_{n,k}^{m} = \begin{cases} q_{n,k}^{m} + q_{n,k}^{m-1} + t_{n,k}^{m}, & n>0 \wedge m \ge 0 \wedge k>0 \wedge k+m \le n \\ 1, & n=0 \wedge m=0 \wedge k=0 \\ 0, & \text{otherwise} \end{cases} \qquad (19)$$

Where $q_{n,k}^{m}$ is defined as:

$$q_{n,k}^{m} = 2 \sum_{j=0}^{n-k-1} \sum_{i=0}^{m} \binom{j}{i} b_j\, p_{n-j-1,k}^{m-i} \qquad (20)$$

The first two subterms in (19/20) exhibit a strong resemblance with (16). The first subterm corresponds to configurations with an unmarked root, the second subterm to those with a marked root. The top-trees contribute the term $t_{n,k}^{m}$. Since $p_{n,k}^{0} \equiv s_{n,k}$, we must have that $p_{0,0}^{0} = 1$. We will proof by generalised induction over n and m (with k fixed) that:

$$p_{n,k}^{m} = \begin{bmatrix} n+m+1 \\ m \end{bmatrix} \begin{bmatrix} 2n-2k \\ n-k-m \end{bmatrix} b_k \,, \quad n > 0 \wedge m \geq 0 \wedge k > 0 \wedge k+m \leq n \tag{21}$$

In section 2.1 we have proved that (21) holds for $m = 0 \vee n = 0$. By the generalised induction principle we may assume that (21) holds for $0 < m' < m$ and $0 < n' < n$. Substitution of (21) in (20) yields:

$$q_{n,k}^{m} = 2\, b_k \sum_{j=0}^{n-k-1} \sum_{i=0}^{m} \begin{bmatrix} j \\ i \end{bmatrix} b_j \begin{bmatrix} n-j+m-i \\ m-i \end{bmatrix} \begin{bmatrix} 2n-2j-2k-2 \\ n-j-k-1-m+i \end{bmatrix}$$

Application of (22), see exercise 31, page 70 in Knuth's book,[8] which states that:

$$\begin{bmatrix} a \\ c \end{bmatrix} \begin{bmatrix} b \\ d \end{bmatrix} = \sum_{x} \begin{bmatrix} c-a+b \\ x \end{bmatrix} \begin{bmatrix} d+a-b \\ d-x \end{bmatrix} \begin{bmatrix} a+x \\ c+d \end{bmatrix} \,, \quad \text{integer } c \geq 0 \,, \text{ integer } d \geq 0 \tag{22}$$

and changing the order of summation yields:

$$q_{n,k}^{m} = 2\, b_k \sum_{x=0}^{k+1} \sum_{j=0}^{n-k-1} \sum_{i=0}^{m} \begin{bmatrix} j \\ i \end{bmatrix} b_j \begin{bmatrix} k+1 \\ x \end{bmatrix} \begin{bmatrix} n-j-2k-2 \\ m-x-i \end{bmatrix} \begin{bmatrix} 2n-2j-2k-2+x \\ n-j-k-1 \end{bmatrix}$$

Application of Vandermonde's convolution to perform the summation over i and substitution of (2) yields:

$$q_{n,k}^{m} = 2\, b_k \sum_{x=0}^{k+1} \sum_{j=0}^{n-k-1} \begin{bmatrix} k+1 \\ x \end{bmatrix} \begin{bmatrix} 2j+1 \\ j \end{bmatrix} \frac{1}{2j+1} \begin{bmatrix} 2n-2j-2k-2+x \\ n-j-k-1 \end{bmatrix} \begin{bmatrix} n-2k-2 \\ m-x \end{bmatrix}$$

Application of Rothe's non-symmetric addition theorem[9] to perform the summation on j yields:

$$q_{n,k}^{m} = 2\, b_k \sum_{x=0}^{k+1} \begin{bmatrix} k+1 \\ x \end{bmatrix} \begin{bmatrix} 2n-2k-1+x \\ n-k-1 \end{bmatrix} \begin{bmatrix} n-2k-2 \\ m-x \end{bmatrix}$$

Using (22) again but in opposite direction to perform the summation on x we obtain:

$$q_{n,k}^{m} = 2\, b_k \begin{bmatrix} 2n-2k-1 \\ n-k-m-1 \end{bmatrix} \begin{bmatrix} n+m+1 \\ m \end{bmatrix}$$

With (18) and (19) it is readily verified that (21) holds for n and m, which concludes the proof of (21).

The total number of subtrees in all permanent marked trees must be equal to the number of unmarked nodes:

$$\sum_{k=1}^{n} p_{n,k}^{m} = \begin{bmatrix} n \\ m \end{bmatrix} (n-m)\, b_n$$

3.8. Average size of permanent marked subtrees

For a uniform probability distribution of trees we have:

$$\overline{p_n^m} = \frac{\sum_{k=0}^{n} k\, p_{n,k}^m}{\begin{bmatrix} n \\ m \end{bmatrix} n\, b_n} = \frac{\sum_{k=0}^{n} k \begin{bmatrix} n+m+1 \\ m \end{bmatrix} \begin{bmatrix} 2n-2k \\ n-k-m \end{bmatrix} b_k}{\begin{bmatrix} n \\ m \end{bmatrix} n\, b_n} \tag{23}$$

Unfortunately, there is no simple general solution for the sum in the enumerator of (23). To see this, let Δ_n^m be defined as:

$$\Delta_n^m = \frac{\sum_{k=0}^{n} k\, p_{n,k}^{m-1}}{\begin{bmatrix} n+m \\ m-1 \end{bmatrix}} - \frac{\sum_{k=0}^{n} k\, p_{n,k}^m}{\begin{bmatrix} n+m+1 \\ m \end{bmatrix}}, \quad 0 < m < n$$

With (21) and m fixed it can be proved by induction on n that:

$$\Delta_n^m = \begin{bmatrix} 2n+1 \\ n-m \end{bmatrix}$$

If the sum of Δ_n^m for all possible values of m is calculated, we find that all but the first and last terms cancel out:

$$\sum_{i=1}^{m} \Delta_n^i = \frac{\sum_{k=0}^{n} k\, p_{n,k}^0}{\begin{bmatrix} n+1 \\ 0 \end{bmatrix}} - \frac{\sum_{k=0}^{n} k\, p_{n,k}^1}{\begin{bmatrix} n+2 \\ 1 \end{bmatrix}} + \frac{\sum_{k=0}^{n} k\, p_{n,k}^1}{\begin{bmatrix} n+2 \\ 1 \end{bmatrix}} - \cdots + \frac{\sum_{k=0}^{n} k\, p_{n,k}^{m-1}}{\begin{bmatrix} n+m \\ m-1 \end{bmatrix}} - \frac{\sum_{k=0}^{n} k\, p_{n,k}^m}{\begin{bmatrix} n+m+1 \\ m \end{bmatrix}}$$

Using this result and the fact that $p_{n,k}^0 = s_{n,k}$ to calculate $\sum_{k=0}^{n} k\, p_{n,k}^m$ we obtain:

$$\sum_{k=0}^{n} k\, p_{n,k}^m = \begin{bmatrix} n+m+1 \\ m \end{bmatrix} \left[\sum_{k=0}^{n} k\, s_{n,k} - \sum_{i=1}^{m} \Delta_n^i \right]$$

$$= \begin{bmatrix} n+m+1 \\ m \end{bmatrix} \left[4^n - \sum_{i=0}^{m} \begin{bmatrix} 2n+1 \\ n-i \end{bmatrix} \right] \tag{24}$$

$$= \begin{bmatrix} n+m+1 \\ m \end{bmatrix} \left[4^n - \sum_{i=0}^{n} \begin{bmatrix} 2n+1 \\ i \end{bmatrix} + \sum_{i=0}^{n-m-1} \begin{bmatrix} 2n+1 \\ i \end{bmatrix} \right]$$

For sums of type $\sum_{k=0}^{n} \begin{bmatrix} m \\ k \end{bmatrix}$, where $n < m$ no simple solution is known (see Knuth's

book, page 64). The best that can be done is to compute $\sum_{k=0}^{n} k\, p_{n,k}^{m}$ for specific values of m. Although this may be done conveniently for small values of m using (24), the method used in the previous sections is employed once more, to show why the problem has no simple solution. For instance let $m = 2$. The generating function for the sequence $\langle \sum_{k=0}^{n} k\, p_{n,k}^{2} \rangle$ with a fixed value of k can be found from $B(z)$ by observing that:

$$\begin{bmatrix} 2n-2k \\ n-k-2 \end{bmatrix} z^{n-k} = \frac{(n-k)(n-k-1)}{(n-k+2)} b_{n-k}\, z^{n-k} = z^2 \frac{d^2}{dz^2} \left[\frac{1}{z^2} \int_0^z \zeta\, b_{n-k}\, \zeta^{n-k}\, d\zeta \right]$$

Hence:

$$\sum_{n=0}^{\infty} \left[\sum_{k=0}^{n} \begin{bmatrix} 2n-2k \\ n-k-2 \end{bmatrix} b_k\, k \right] z^n = z\, B'(z)\, z^2 \frac{d^2}{dz^2} \left[\frac{1}{z^2} \int_0^z \zeta\, B(\zeta)\, d\zeta \right]$$

$$= \frac{1}{1-4z} + \frac{z-1}{2z^2\sqrt{1-4z}} - \frac{(2z+1)\sqrt{1-4z}}{4z^3} - \frac{(1-4z)^{1\frac{1}{2}}}{4z^3} + \frac{1}{2z^3} - \frac{1}{z^2} + \frac{1}{2z} \qquad (25)$$

$$= \sum_{n=0}^{\infty} \left[(n+2)(n+3)\, 4^n - (2n+1)(3n^2+7n+6)\, b_n \right] z^n$$

To calculate $\overline{p_n^m}$, the generating function for the sequence $\langle \begin{bmatrix} 2n-2k \\ n-k-m \end{bmatrix} \rangle$ has to be found, with k and m fixed. This requires $m-1$ integrations, followed by m differentiation operations. Comparing (14) and (25) we may assert, that the structure of the function $B(z)$ does not permit a simple generalisation to such a procedure.

3.9. Approximation and numerical data

The values of $\overline{p_n^m}$ for large n and comparatively small r are calculated by deriving an approximation for a combination of (23) and (24):

$$\overline{p_n^m} = \frac{\begin{bmatrix} n+m+1 \\ m \end{bmatrix}}{\begin{bmatrix} n \\ m \end{bmatrix}} \left[\frac{4^n}{n\, b_n} - \sum_{i=0}^{m} \frac{\begin{bmatrix} 2n+1 \\ n-i \end{bmatrix}}{n\, b_n} \right] \qquad (26)$$

The quotients of binomial coefficients in this equation can be written as quotients of polynomials in n with integer coefficients:

$$\frac{\binom{2n+1}{n-i}}{n\,b_n} = \frac{(2n+1)}{n}\,\frac{n\,(n-1)\,\cdots\,(n-i+1)}{(n+2)\,(n+3)\,\cdots\,(n+i+1)}$$

$$= \frac{(2n+1)}{n}\,\frac{u_i(n)}{v_i(n)}\,, \quad \mathrm{lead}(u_i) = \mathrm{lead}(v_i) = 1 \wedge \deg(u_i) = \deg(v_i)$$

$$\frac{\binom{n+m+1}{m}}{\binom{n}{m}} = \frac{(n+m+1)\,(n+m)\,\cdots\,(n+2)}{n\,(n-1)\,\cdots\,(n-m+1)}$$

$$= \frac{u_{m+1}(n)}{v_{m+1}(n)}\,, \quad \mathrm{lead}(u_{m+1}) = \mathrm{lead}(v_{m+1}) = 1 \wedge \deg(u_{m+1}) = \deg(v_{m+1})$$

For any two polynomials $u(n)$ and $v(n)$ over the field of rational numbers, there exists a unique pair of polynomials $q(n)$ and $r(n)$ such that:[10]

$$u(n) = v(n) \times q(n) + r(n)\,, \quad \deg(r) < \deg(v)$$

Since $\forall\, i \in 0..m+1$ the leading coefficients of $u_i(n)$ and $v_i(n)$ are 1 and the degrees of both polynomials are equal we have $q_i(n) \equiv 1$.

$$\overline{p_n^m} = \left(1 + \frac{r_{m+1}(n)}{v_{m+1}(n)}\right)\left(\frac{4^n}{n\,b_n} - \frac{(2n+1)}{n}\sum_{i=0}^{m}\left(1 + \frac{r_i(n)}{v_i(n)}\right)\right)$$

$$= \sqrt{\pi\, n} - 2(m+1) + \mathrm{O}\left(\frac{1}{\sqrt{n}}\right)\,, \quad n \gg m$$

Table (2) shows some values of $\overline{p_n^m}$ for small n and m, which where calculated using (26). The last line in the table gives approximated values using the formula $\sqrt{8192\,\pi} - 2(m+1)$. This shows that for small m the approximation is good, but for m near $\sqrt{n}$ it has no significance.

n	m=0	m=1	m=2	m=4	m=8	m=16	m=32	m=64
2	1.50	.50						
4	2.32	1.23	.63					
8	3.60	2.38	1.61	.74				
16	5.53	4.16	3.20	2.00	.84			
32	8.35	6.84	5.69	4.09	2.34	.91		
64	12.41	10.79	9.46	7.43	4.94	2.60	.95	
128	18.22	16.50	15.01	12.59	9.25	5.65	2.77	.97
256	26.48	24.68	23.07	20.29	16.09	10.94	6.19	2.88
512	38.19	36.34	34.62	31.54	26.55	19.69	12.36	6.54
1024	54.78	52.89	51.09	47.78	42.08	33.51	23.04	13.41
2048	78.26	76.33	74.48	70.98	64.72	54.56	40.65	25.83
4096	111.47	109.52	107.63	103.99	97.28	85.75	68.41	47.28
8192	158.45	156.48	154.56	150.82	143.76	131.12	110.63	82.60
app.	158.42	156.42	154.42	150.42	142.42	126.42	94.42	30.42

Table 2 : $\overline{p_n^m}$ for some values of n and m.

4. Conclusions

The practical importance of graph reduction has provided the incentive to develop a model for the effects that reduction has on the structure of its graphs. The question that we have worked on in this paper is: how many nodes may be expected to become unreachable when an arbitrary edge is cut? In doing so a method has been developed that makes some of the standard results of Catalan statistics applicable to graphs. The method works by treating a shared node in a graph as a specially marked node in a tree. The standard enumeration method for ordered binary trees is extended to exclude subtrees with marked nodes. Disconnecting a single edge in a graph may cause shared nodes to become unreachable, because all paths to such shared nodes pass via that single edge. Assuming that all shared nodes are from this particular (removable) type, an upper bound on the expected size of a subtree is calculated ($\overline{r_n^m}$). An estimate from below is obtained by assuming that all shared nodes remain connected (permanent) if an arbitrary edge is cut. The average size of a subtree under this assumption is also derived ($\overline{p_n^m}$).

It turns out that in both cases the expected number of nodes that become unreachable when an arbitrary edge is cut is larger than we have observed in practice. One reason may be that not all edges have the same probability of being cut. Reduction probably takes place more near the leaves than near the root of the graph. However, without further investigation we can only speculate on possible probability distributions. Although the formulae that we have derived are specific with respect to the uniform distribution that we have assumed, the method allows for other distributions to be used.

Acknowledgements

I am grateful to Henk Barendregt and Arthur Veen for comments on an earlier draft of the paper. Peter van Emde Boas and Michiel Smid helped me sort out the literature.

References

1. H. P. Barendregt, *The lambda calculus, its syntax and semantics,* North Holland, Amsterdam (1984).
2. P. J. Landin, ''The mechanical evaluation of expressions,'' *Computer Journal* **6**(4) pp. 308-320 (Jan. 1964).
3. C. P. Wadsworth, *Semantics and pragmatics of the lambda calculus,* Oxford Univ, U.K. (1971). PhD. Thesis
4. P. H. Hartel, ''A comparative study of three garbage collection algorithms,'' PRM project internal report D-23, Dept. of Comp. Sys, Univ. of Amsterdam (Feb. 1988).
5. J. Cohen, ''Garbage collection of linked structures,'' *Computing Surveys* **13**(3) pp. 341-367 (Sep. 1981).
6. N. G. de Bruijn, D. E. Knuth, and S. O. Rice, ''The average height of planted plane trees,'' pp. 15-22 in *Graph theory and computing*, ed. R. C. Read,Academic Press, London, U.K. (1972).
7. P. H. Hartel and A. H. Veen, ''Statistics on graph reduction of SASL programs,'' *Software practice and experience* **18**(3) pp. 239-253 (Mar. 1988).
8. D. E. Knuth, *The art of computer programming, volume 1: Fundamental algorithms,* Addison Wesley, Reading, Massachusetts (1973). second edition
9. H. W. Gould and J. Kaucky, ''Evaluation of a class of binomial coefficient summations,'' *Journal of Combinatorial theory* **1**(2) pp. 233-247 (Sep. 1966).
10. D. E. Knuth, *The art of computer programming, volume 2: Seminumerical algorithms,* Addison Wesley, Reading, Massachusetts (1980). second edition

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

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 pointer 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 denounced 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. The amount of work involved in normalising a job outweighs its communication cost.

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.

---

† 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:

$$\forall_{i \in 1..n} \left[c_i < \sum_{k=1, k \neq i}^{n} s_k \right] \tag{1}$$

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: $\forall_{i \in 1..n} \left[c_i + s_i < \sum_{k=1}^{n} s_k \right]$ and therefore (2).

The intuitive version of job condition (3), namely $\forall_{i \in 1..n} \; c_i < s_i$ 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.

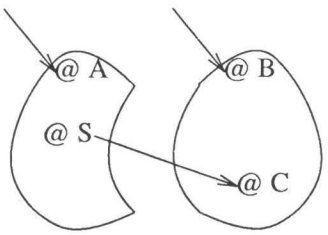


Figure 1 : An external pointer

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).

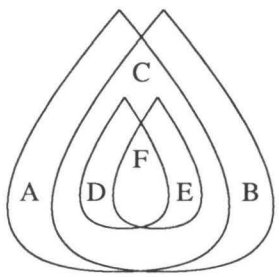


Figure 2 : Nested sharing

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. 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 six possible ways to annotate 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 $(F\ x_1\ x_2\ \cdots\ x_n)$ with the following restrictions (the sandwich conditions):

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

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

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

---

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

In the framework the SASL programming language[18] a new primitive function (see figure 3) has been introduced, which supports the sandwich strategy. We have used the device due to Landin rather than Currying to support a variable number of arguments. The latter method requires extra parameters to indicate the number of arguments to F and the G_i. By using a list, the number of arguments can be found by counting its length.

$$sandwich\ F\ ((G_1, a_{11}, \cdots, a_{1k_1}), \cdots, (G_n, a_{n1}, \cdots, a_{nk_n})) \rightarrow$$

$$F\ (G_1\ a_{11}\ \cdots\ a_{1k_1})\ \cdots\ (G_n\ a_{n1}\ \cdots\ a_{nk_n})$$

Figure 3 : Rewrite rule for the sandwich function

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

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] which makes it harder to devise good heuristics.

4. Application of the sandwich strategy

As a first example of the sandwich transformation of a divide-and-conquer problem we consider the quick sort algorithm. The principle of this transformation also applies to other divide-and-conquer algorithms, as will be shown by the sandwich 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 (4) shows the essential part of the quick sort algorithm in SASL:

$$
\begin{aligned}
&\mathit{QuickSort}\ (a : b) = \mathit{append}\ (\mathit{QuickSort}\ m)\ (\mathit{cons}\ a\ (\mathit{QuickSort}\ n)) \\
&\qquad\qquad \text{WHERE} \\
&\qquad\qquad m : n = \mathit{split}\ a\ b
\end{aligned}
$$

Figure 4 : Quick sort

The application of *append* is similar to a sandwich expression, but lacks an essential property. The second argument to *append* is an application of *cons* which does not satisfy sandwich condition (5), since ($QuickSort\ n$) is not a normal form. If both applications of *QuickSort* were to be reduced in parallel, the application ($split\ a\ b$) 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 ($split\ a\ b$) before the creation of the jobs. Since the code for *QuickSort* is already in normal form, sandwich condition (5) is now satisfied. The sandwich version of the *QuickSort* program is shown in figure (5).

```
QuickSort (a : b) = sandwich' append ((QuickSort , m), (QuickSort , n))
                    WHERE
                    append l r = l ++ (a : r)
                    m : n = split a b
```

Figure 5 : The sandwich version of quick sort

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

```
Threshold = 100

PlainQuickSort () = ()
PlainQuickSort (a : x) = PlainQuickSort m ++ (a : PlainQuickSort n)
                         WHERE
                         m , n = PlainSplit a x () ()

PlainSplit a () m n = m , n
PlainSplit a (b : x) m n = b < a → PlainSplit a x (b : m) n
                           PlainSplit a x m (b : n)

QuickSort () = ()
QuickSort (a : x) = lm > Threshold →
                        ln > Threshold →
                            sandwich append ((QuickSort , m), (QuickSort , n))
                        QuickSort m ++ (a : PlainQuickSort n)
                    ln > Threshold → PlainQuickSort m ++ (a : QuickSort n)
                    PlainQuickSort m ++ (a : PlainQuickSort n)
                    WHERE
                    m , lm , n , ln = Split a x () 0 () 0
                    append l r = l ++ (a : r)
```

$$
\begin{aligned}
&Split\ a\ ()\ m\ lm\ n\ ln = m\ ,\ lm\ ,\ n\ ,\ ln \\
&Split\ a\ (b\ :x)\ m\ lm\ n\ ln = b < a \rightarrow Split\ a\ x\ (b\ :m)\ (lm+1)\ n\ ln \\
&\qquad\qquad\qquad\qquad\qquad\qquad Split\ a\ x\ m\ lm\ (b\ :n)\ (ln+1)
\end{aligned}
$$

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

The cost involved in the control mechanism 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 the sequel to this paper.

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,[20] 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 sandwich version of the program is shown in figure (7). We have to call upon *sandwich'* again to guarantee, that the input list is split before any jobs are created.

$$
\begin{aligned}
&fft\ 1\ r\ \vec{d} = \vec{d} \\
&fft\ n\ r\ \vec{d} = sandwich'\ append\ \ ((fft\ ,\ halfn\ ,\ halfr\ ,\ \vec{u}),\ (fft\ ,\ halfn\ ,\ (halfr\ +\ 128),\ \vec{v})) \\
&\qquad\qquad \text{WHERE} \\
&\qquad\qquad append\ \vec{p}\ \vec{q} = \vec{p}\ +\!+\ \vec{q} \\
&\qquad\qquad halfn\ = n\ /\ 2 \\
&\qquad\qquad halfr\ = r\ /\ 2 \\
&\qquad\qquad \vec{u} = \vec{x} + \vec{z} \\
&\qquad\qquad \vec{v} = \vec{x} - \vec{z} \\
&\qquad\qquad \vec{x}\ ,\ \vec{y} = split\ \vec{d}\ halfn \\
&\qquad\qquad \vec{z} = \vec{y} * \exp\ (halfr\ *\ i\ *\ \pi / 128)
\end{aligned}
$$

Figure 7 : The 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, no recoding is necessary to provide this information 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 is therefore straight forward.

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.[21] According to Michielse and van der Vorst[22] 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.

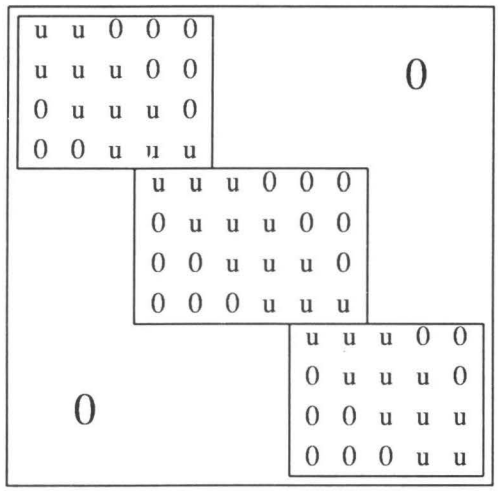


Figure 8 : Partitioning of a tri-diagonal matrix ($u \neq 0$)

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

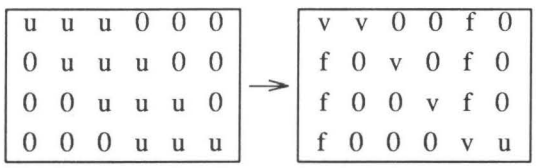


Figure 9 : First elimination in one block

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).

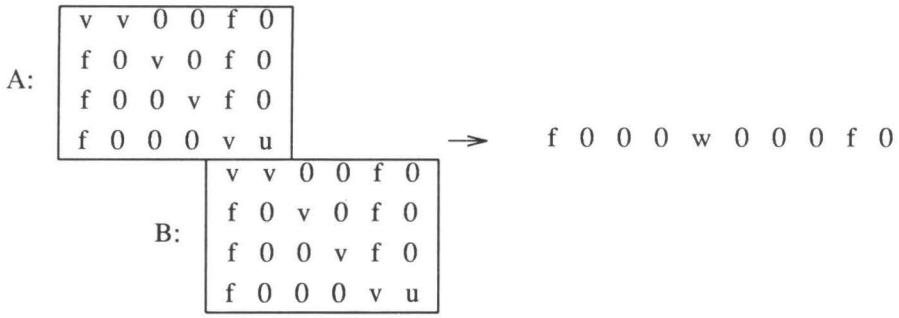


Figure 10 : Elimination at the borders of the blocks

The rightmost column containing the fill-in of matrix A is the same column as the left-most 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.

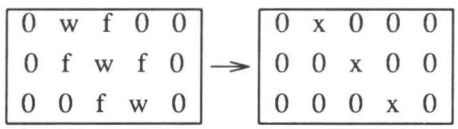


Figure 11 : Elimination of the subsystem

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.

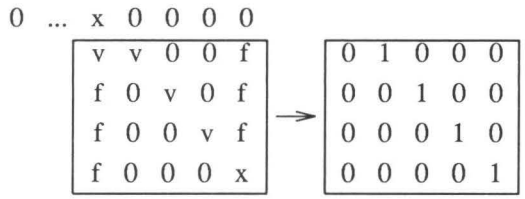


Figure 12 : Final elimination

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

$$
\begin{aligned}
&Partition\ matrix = ParMap\ SecondElimination\ matrix_2 \\
&\qquad\qquad \text{WHERE} \\
&\qquad\qquad matrix_2 = SequentialPart\ matrix_1 \\
&\qquad\qquad matrix_1 = ParMap\ FirstElimination\ matrix \\
&ParMap\ f\ (a : ()) = (f\ a) : () \\
&ParMap\ f\ (a : x) = sandwich'\ cons\ ((f, a), ((ParMap\ f), x))
\end{aligned}
$$

Figure 13 : Skeleton of Wang's algorithm in SASL

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.

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 G_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' F ((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)
F (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' F* ((*remote* , 0 , 0 , 2) ,)		
		3	*remote* 0 0 2
		4	*H* 0 0
		5	(*own* 10) : 7
6	*F* (*"remote name"* : 7)		
7	*repeat "remote name"* 14 2		
8	*sandwich' F* ((*remote* , *"remote name"* , 14 , 1) ,)		
		9	*remote* 10 14 1
		10	*H* 10 14
11	*F* (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 *F* (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 *F* has produced its result.

5.2. A parallel hydraulical simulation

A functional program that implements a mathematical model of the tides in the North Sea[23] 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.[24] 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 are updated in parallel.

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

$$repeat\ f\ x\ 0 = GetRemoteData\ x$$
$$repeat\ f\ x\ n = repeat\ f\ (f\ x)\ (n-1)$$

Figure 16 : The main recursion.

The function *Update* submits the matrices *Left* and *Right* to different processors, where the actual updating takes place in parallel. All subsequent recursive invocations of *Update* 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*. 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.

```
Update (Left : (Right : BorderOfRight))
    = sandwich' cons ((UpdateYHleft, Left'), (UpdateYHright, Right', BorderOfLeft'))
      WHERE
      (Left' : BorderOfLeft') : Right'
          = sandwich' cons ((UpdateXleft, Left, BorderOfRight), (UpdateXright, Right))
```

Figure 17 : The two phase updating.

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

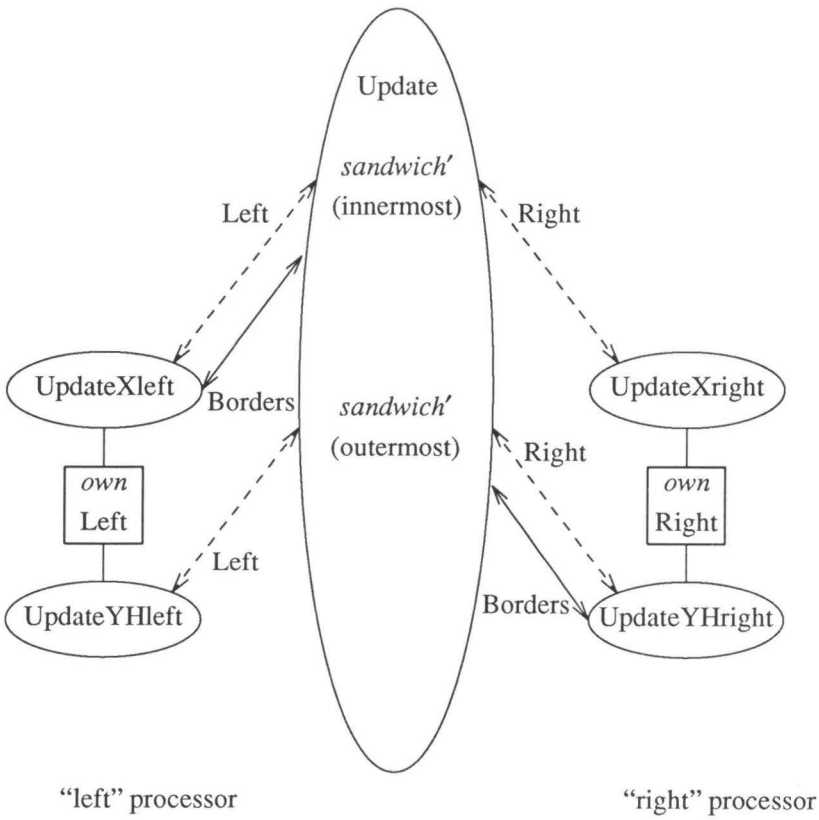


Figure 18 : Communication structure of the tidal model

The normalising variant of the sandwich has to be used in this program, to obtain the

correct sequence of both updates. Once the evaluator requires the result of the function *Update* (see figure 17), reduction continues with the execution of the outermost sandwich application. The normalising property of the latter function will require the evaluation of the arguments: *Left'*, *Right'* and *BorderOfLeft'*. This evaluation in turn forces the execution of the innermost sandwich application. So the innermost sandwich will execute prior to the outermost sandwich. The updating of the x-velocities will run in parallel, yielding the normal forms *Left'*, *Right'* and *BorderOfLeft'*, 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'* is returned and passed unmodified as an argument to *UpdateYHleft*. The updating of matrix *Right* 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'* could be retained in this processor and a remote name could be returned instead. The same applies to the matrix *Right'* and the functions *UpdateXright* and *UpdateYHright*. The only real data to be returned and retransmitted is the *BorderOfLeft'*, which travels from the "left" processor to the "right" processor. Figure (19) shows the annotation that is necessary to obtain the desired behaviour:

$$\begin{aligned} UpdateXleft\ Left\ BorderOfRight &= (own\ Left') : RightBorderOf\ Left' \\ &\quad \text{WHERE} \\ &\quad Left' = updateXleft\ Left\ BorderOfRight \end{aligned}$$

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

Figure 19 : Retention of the left matrix.

The function *UpdateXleft* returns a remote name for matrix *Left'* and real data for the border of *Left'*. The actual updating takes place in the function *updateXleft* (without capital U). *UpdateXright* just returns a remote name for matrix *Right'*. Both functions retain the actual matrices in the processors they have been assigned to by the sandwich. Because the remote names *Left'* and *Right'* 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'* and *Right'* 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* 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'*:

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

$$UpdateYHright\ Right\ BorderOfLeft = (own\ Right') : (LeftBorderOf\ Right')$$
$$\text{WHERE}$$
$$Right' = updateYHright\ Right\ BorderOfLeft$$

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):

$$GetRemoteData\ (Left : Right) = sandwich'\ cons\ ((I\ , Left)\ , (I\ , Right))$$

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 sandwich of figure (21) will therefore submit two jobs, being the application of the identity function to *Left* and *Right*, 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 *cons* function pairs the two matrices, representing the state of the tidal model after n iterations.

6. 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,[25] 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[26] 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.[27] The way a closure is implemented brings about some locality.

The "serial" combinator[8,28] 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[29] 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]

7. 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 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 will be pursued in the sequel to this paper.

Acknowledgements

We greatfully 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," *International journal of 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," *International journal of computer and information sciences* **8**(6) pp. 435-471 (Dec. 1979).
4. P. C. Treleaven and R. P. Hopkins, "A recursive computer for VLSI," *9-th IEEE/ACM symp. on computer architecture, Computer architecture news* **10**(3) pp. 229-238 (Apr. 1982).
5. M. Amamiya, "A new parallel graph reduction model and its machine architecture," pp. 329-356 in *France-Japan artificial intelligence and computer science symp.*, North Holland (1986).
6. F. W. Burton and M. R. Sleep, "Executing functional programs on a virtual tree of processors," pp. 187-194 in *Conf. on functional programming languages and computer architecture*, ed. Arvind,ACM, 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 *Conf. on functional programming languages and computer architecture*, ed. Arvind,ACM, Portsmouth, New Hampshire (Oct. 1981).
8. P. Hudak and B. 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 multiprocessing system," pp. 613-622 in *National computer conf.*, ed. R. E. Merwin, 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 *Third conf. on functional programming languages and computer architecture, LNCS 274*, ed. G. Kahn,Springer verlag, 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 (Sep. 1984).

14. T. Johnsson, "Efficient compilation of lazy evaluation," *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 *Third conf. on functional programming languages and computer architecture, LNCS 274*, ed. G. Kahn,Springer verlag, Portland, Oregon (Sep. 1987).

16. J. Fairbairn and S. Wray, "Tim: A simple lazy abstract machine to execute super-combinators," pp. 34-45 in *Third conf. on functional programming languages and computer architecture, LNCS 274*, ed. G. Kahn,Springer verlag, 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, J. C. Shepherdson,Prentice Hall, London (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 dataflow machine," pp. 1-15 in *Third conf. on functional programming languages and computer architecture, LNCS 274*, ed. G. Kahn,Springer verlag, Portland, Oregon (Sep. 1987).

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

21. H. H. Wang, "A parallel method for tri-diagonal equations," *ACM transactions on Mathematical Software* **7**(2) pp. 170-183 (Jun. 1981).

22. P. H. Michielse and H. A. van der Vorst, "Data transport in Wang's partition method," Internal report 86-32, Dept. of Comp. Sci, Technical Univ. Delft (1986).

23. W. G. Vree, "The grain size of parallel computations in a functional program," pp. 363-370 in *Conf. on Parallel processing and Applications*, ed. E. Chiricozzi, A. d'Amico,Elsevier Science Publishing, L'Aquila, Italy (Sep. 1987).

24. W. G. Vree, "Parallel graph reduction for communicating sequential processes," PRM project internal report D-26, Dept. of Comp. Sys, Univ. of Amsterdam (Aug. 1988).

25. 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).

26. D. L. McBurney and M. R. Sleep, ''Transputer based experiments with the ZAPP architecture,'' pp. 242-259 in *Parle conf. on parallel architectires and languages, LNCS 259*, ed. J. W. de Bakker, A. J. Nijman, P. C. Treleaven,, Eindhoven, The Netherlands (Jun. 1987).

27. R. M. Keller, ''Distributed graph reduction from first principles,'' Technical report, Dept. of Comp. Sci, Univ. of Utah (1985).

28. P. Hudak and B. Goldberg, ''Serial combinators: "Optimal" grains of parallelism,'' pp. 382-399 in *Second conf. on functional programming languages and computer architecture, LNCS 201*, ed. J.-P. Jouannaud,Springer verlag, Nancy, France (Sep. 1985).

29. R. J. M. Hughes, ''Super combinators - A new implementation method for applicative languages,'' pp. 1-10 in *ACM symp. on Lisp and functional programming*, ACM, Pittsburg, Pennsylvania (Aug. 1982).

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, that can be implemented efficiently on an architecture without a common store. In our previous 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. An annotation is available to normalise shared subexpressions before the generation of parallel jobs.

Jobs arise when a special combinator "sandwich" is encountered during the evaluation of an application. The combinator can be viewed as an annotation of a function application. It gives the strict 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 the current 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.

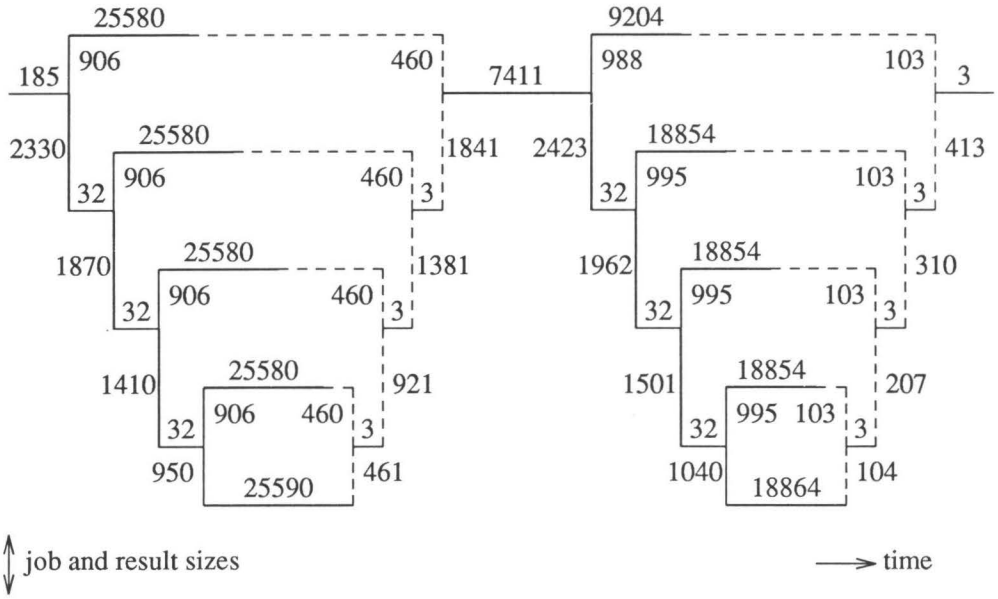


Figure 1 : The job and result trees generated by Wang's algorithm (not drawn to scale)

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 by far from optimal, since only the processing elements involved with the leaf nodes in the job-tree are active. 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 our previous 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 (after the list structure in the sandwich expression has been turned into the appropriate apply structure).

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 one of the pointer fields of the original 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.

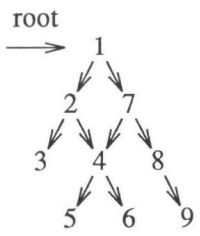


Figure 3 : (a) sample graph

Initial stack configuration
1 2 3
1 2 4 5
1 2 4 6
1 2 4
1 2 → stack growth
1 7 4*
1 7 8 9
1 7 8
1 7
1
Final stack configuration

(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.[6] The same reasoning also holds for mutually recursive functions.

The solution to the Hamming problem[7] 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.[8] 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.[9] 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.[10]

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.

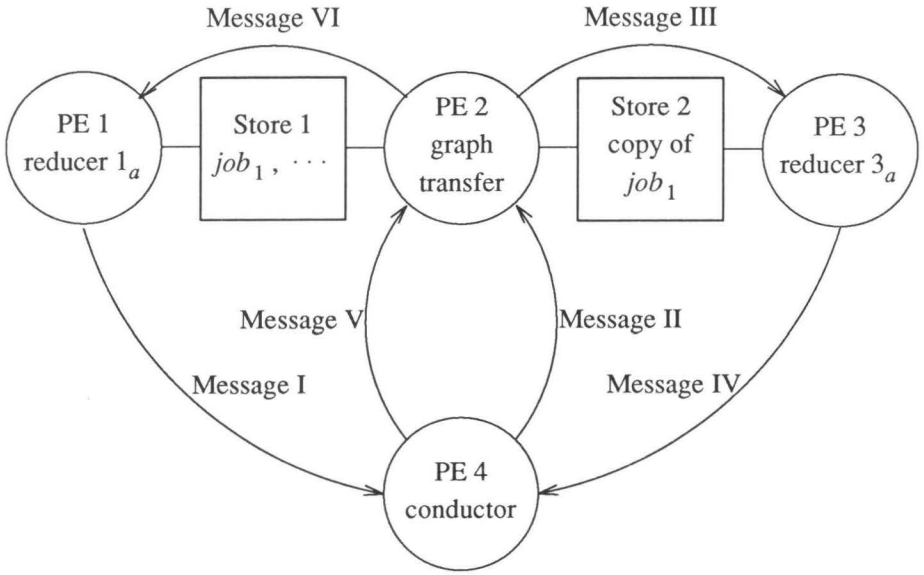


Figure 4 : Graph and message transport

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 multiprogramming 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.[11] 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.

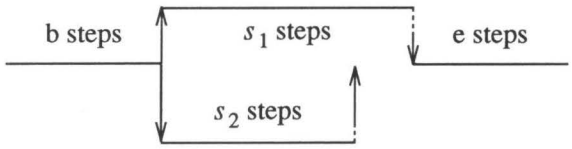


Figure 5 : Time diagram of parallel evaluation

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).

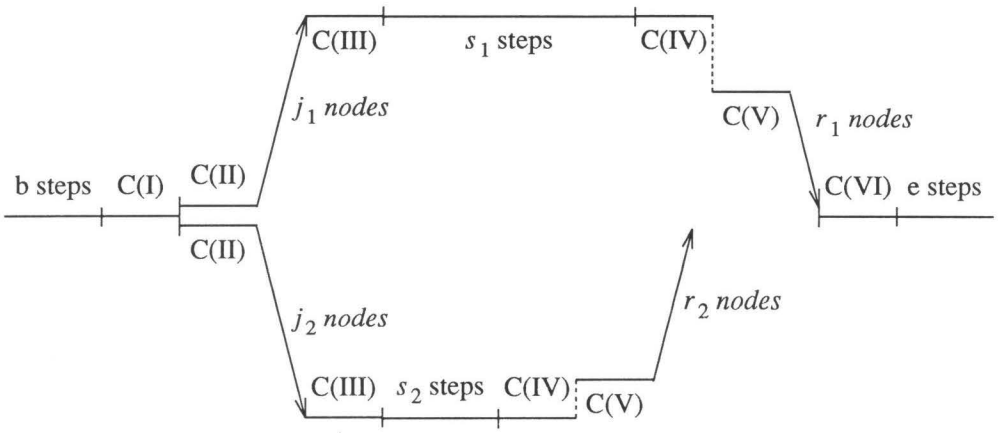


Figure 6 : Time diagram of parallel evaluation and transportation

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).

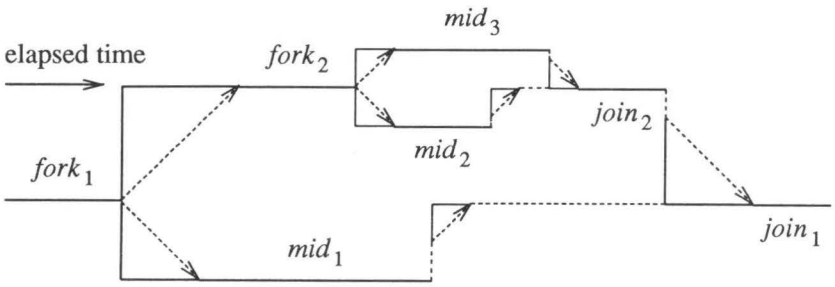


Figure 7 : Fork, mid and join jobs

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.

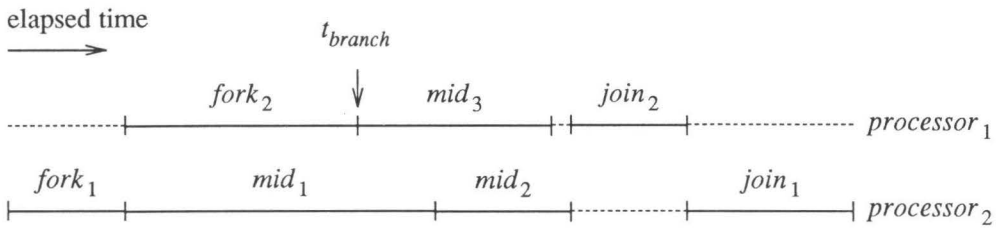


Figure 8 : An optimal schedule with two processors

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.[12] 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.e. 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.[13]

With bound and heuristics the scheduling algorithm performs about twice as fast as the complete search.

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 is such that all mid jobs are approximately of the same size.

The SASL function *Alloc* of figure (9) implements the tree search algorithm without the lower bound calculation and cancelling of subtrees. It shows how the sandwich annotation is used in combination with a threshold mechanism.

```
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)
```

Figure 9 : 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.[6] 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.[11] 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,[14] 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. Tuynman & Hertzberger[15] 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* × *seconds* = 10000×0.002 = 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 one of this publication. Particular attention has been paid to annotation and transformation to adopt the applications to parallel execution. In this part of the publication 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.

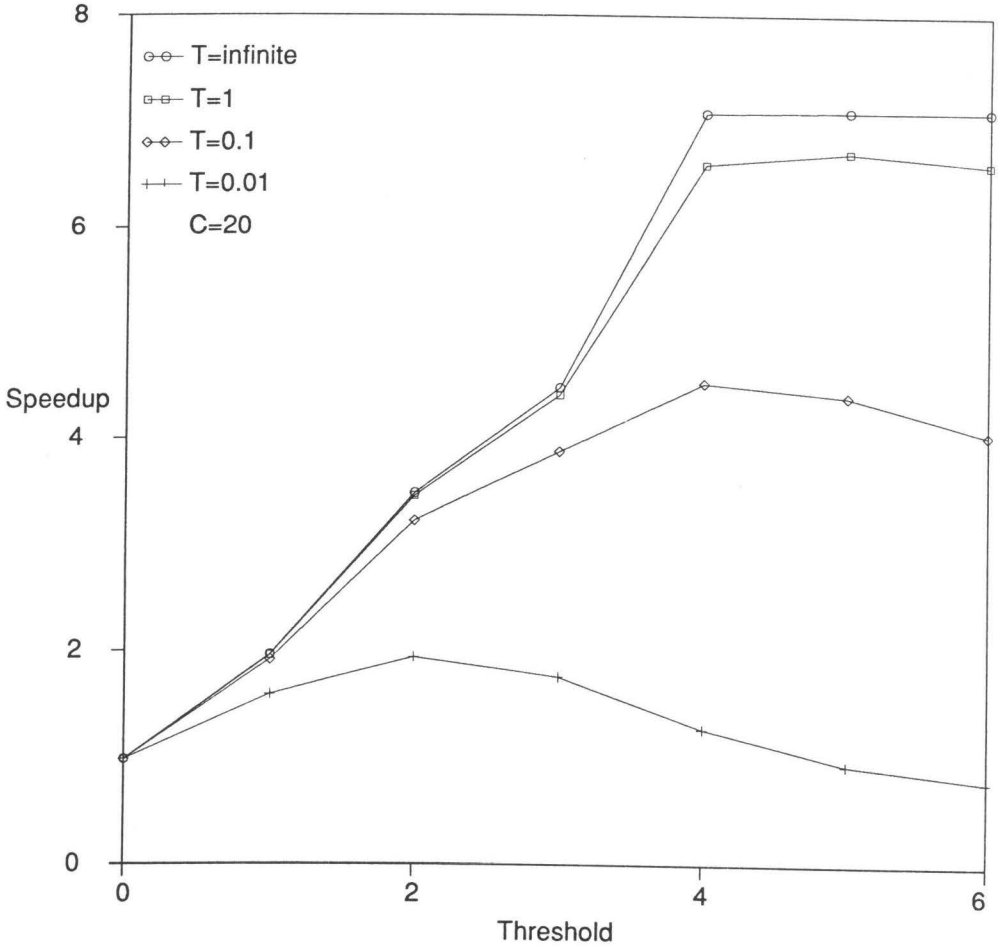


Figure 10 : Speedup of the scheduling program for 8 processors with various T-factors

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 appears to have a critical region between $T = 1$ and $T = 0.1$.

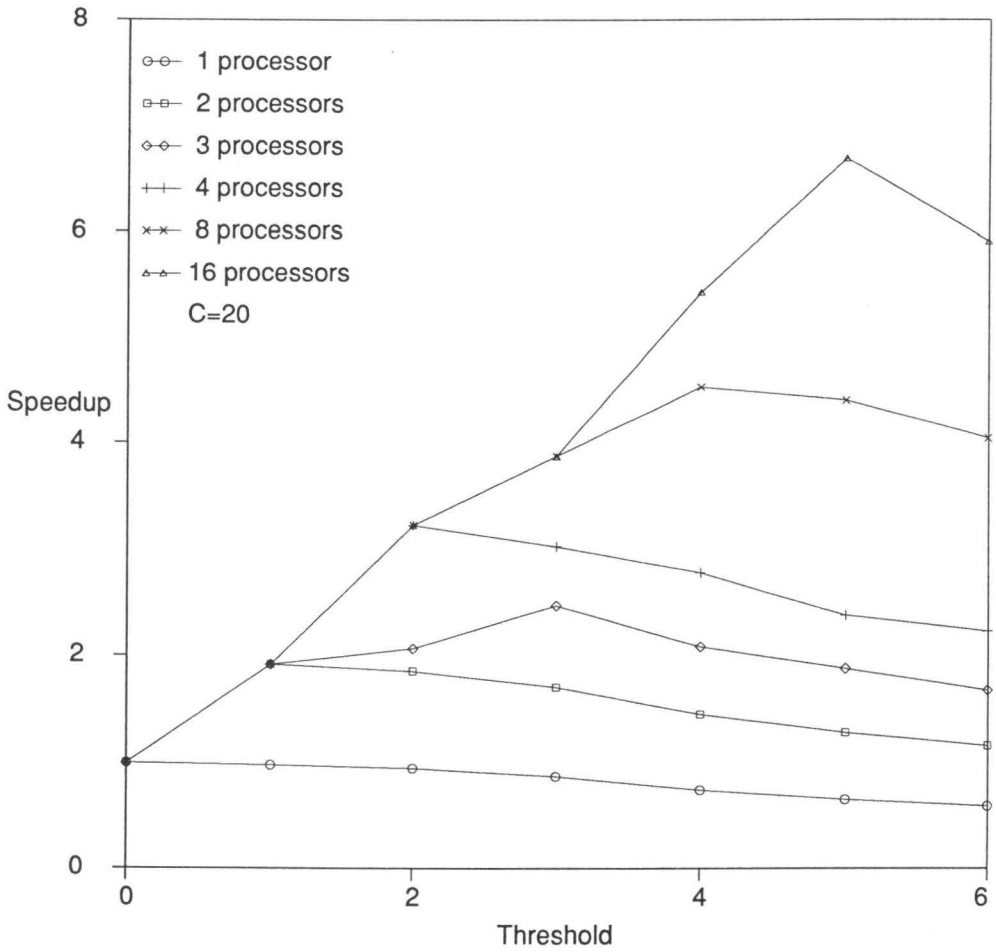


Figure 11 : Speedup of scheduling for $T = 0.1$ with various numbers of processors

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* annotations 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. 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 greatfully 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 D-15, Dept. of Comp. Sys, Univ. of Amsterdam (Apr. 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 D-18, Dept. of Comp. Sys, Univ. of Amsterdam (May. 1988).
4. J. Cohen, "Garbage collection of linked structures," *Computing Surveys* **13**(3) pp. 341-367 (Sep. 1981).
5. P. H. Hartel, "A comparative study of three garbage collection algorithms," PRM project internal report D-23, Dept. of Comp. Sys, Univ. of Amsterdam (Feb. 1988).
6. P. H. Hartel, *Performance analysis of storage management in combinator graph reduction,* Dept. of Comp. Sys, Univ. of Amsterdam (Oct. 1988). PhD. Thesis

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

8. W. G. Vree, ''Parallel graph reduction for communicating sequential processes,'' PRM project internal report D-26, Dept. of Comp. Sys, Univ. of Amsterdam (Aug. 1988).

9. P. H. Hartel and A. H. Veen, ''Statistics on graph reduction of SASL programs,'' *Software practice and experience* **18**(3) pp. 239-253 (Mar. 1988).

10. R. B. Kieburtz, ''The G-machine: A fast, graph-reduction evaluator,'' pp. 400-413 in *Second conf. on functional programming languages and computer architecture, LNCS 201*, ed. J.-P. Jouannaud,Springer verlag, Nancy, France (Sep. 1985).

11. D. A. Turner, ''A new implementation technique for applicative languages,'' *Software Practice and Experience* **9**(1) pp. 31-49 (Jan. 1979).

12. N. J. Nilsson, *Problem solving methods in artificial intelligence,* McGraw-Hill (1971).

13. R. L. Graham, ''Bounds for certain multiprocessing anomalies,'' *Bell system technical journal* **45**(9) pp. 1563-1581 (Nov. 1966).

14. P. H. Hartel, ''Performance of compiled fixed combinator graph reduction,'' PRM project internal report D-27, Dept. of Comp. Sys, Univ. of Amsterdam (Aug. 1988).

15. F. Tuynman and L. O. Hertzberger, ''A distributed real-time operating system,'' *Software practice and experience* **16**(5) pp. 425-441 (May. 1986).

Samenvatting

Een gangbare manier om een functionele programmeertaal te implementeren is door middel van graafreductie. Een functioneel applicatieprogramma wordt voorgesteld door een graaf, die in een aantal stappen wordt getransformeerd tot het resultaat van de applicatie. Afhankelijk van het gebruikte reductiesysteem, wordt de graaf tijdens elke reductiestap ook daadwerkelijk getransformeerd of worden tussenresultaten via een stack berekend. Geheugenbeheer speelt een belangrijke rol bij de implementatie van functionele programmeertalen. Waar bijvoorbeeld in een niet functionele taal als Pascal van een stack gebruik wordt gemaakt om (tussen-) resultaten op te slaan, kan in een functionele taal vaak slechts de heap worden benut. Het parametermechanisme dat in functionele talen veel wordt gebruikt (call by name), laat zich niet zonder meer met behulp van een stack implementeren.

Vanwege de specifieke "first in last out" manier waarop een stack gebruikt wordt, bestaan daarbij geen principiële problemen met betrekking tot geheugenbeheer. Het kan hoogstens voorkomen dat de stack te groot wordt om in het beschikbare geheugen te passen, maar omdat de gehele stack bewaard dient te worden, moet in zo'n geval de verwerking worden afgebroken omdat er te weinig geheugen aanwezig is. De ruimte die in beslag wordt genomen door gegevens die niet langer nodig zijn, wordt direct opnieuw gebruikt. Het beheren van een heap, waaruit vele duizenden cellen per seconde worden aangevraagd en weer worden vrijgegeven, is een lastiger probleem. Niet omdat er geen geschikte algoritmen zouden zijn om de heap te beheren, maar omdat het beslag dat een beheersmechanisme op de verwerkingscapaciteit legt, veelal onaanvaardbaar groot is. Een efficiënte implementatie van een functionele programmeertaal dient slechts een gering deel (enkele procenten) van die capaciteit te besteden aan geheugenbeheer. Ook indien een aparte verwerkingseenheid beschikbaar is om het geheugenbeheer te verzorgen, zal door de noodzaak tot interactie tussen de aanvrager en de beheerder van cellen, de aanvrager aanzienlijke vertraging oplopen.

Geheugenbeheer omvat aspecten die elk verantwoordelijk zijn voor een gedeelte van de belasting. Het aanvragen van nieuwe cellen kan meestal op efficiënte wijze geschieden, door bijvoorbeeld een lijst van vrije cellen bij te houden. Veel tijdrovender is het de cellen te ontdekken die geen deel meer uitmaken van de representatie van het functionele applicatieprogramma. Dit aspect van geheugenbeheer wordt in het algemeen aangeduid als "garbage collection". Het grote aantal garbage collection methoden kan in twee catagoriën worden onderverdeeld. De eerste, de "reference counting" methode houdt van elke geheugencel bij, hoeveel referenties er nog naar die

cel bestaan. Een cel kan opnieuw worden gebruikt, zodra dit aantal tot nul is gedaald. Deze methode is minder geschikt indien ketens van referenties uiteindelijk weer bij de cel terecht komen waar ze van uit zijn gegaan. Een graaf met dergelijke ketens is cyclisch. De tweede categorie garbage collection methoden doorlopen af en toe de graaf, zodat door eliminatie is vast te stellen welke cellen niet langer in gebruik zijn. Vaak wordt de garbage collection pas uitgevoerd als alle beschikbare geheugencellen zijn toegewezen. Sommige systemen zijn dan minuten lang uitsluitend bezig met garbage collection, hetgeen voor de gebruikers van zo'n systeem verre van comfortabel is. Met reference counting worden cellen direct weer opnieuw gebruikt, terwijl de overige methoden het hergebruik uitstellen tot een volgende actie van de garbage collector. Reference counting spreidt van nature de voor geheugenbeheer benodigde verwerkingscapaciteit beter. Een derde aspect van geheugenbeheer betreft de vrijheid welke in het algemeen bestaat bij het toewijzen van nieuw aangevraagde cellen. Het is mogelijk door geschikte keuze localiteit in de representatie van een functioneel programma te bevorderen, teneinde daarmee de prestaties van het systeem als geheel te verbeteren. Dit laatste aspect valt buiten het kader van deze verhandeling.

In dit proefschrift houden we ons in hoofdzaak bezig met garbage collection problematiek, toegespitst op een graafreductiesysteem dat combinators gebruikt voor de overdracht van functie-argumenten. Een combinator is zelf een functie met een aantal argumenten die dient om die argumenten op een bepaalde wijze met elkaar te combineren. In de hoofdstukken in het eerste deel van het proefschrift wordt ingegaan op deze methode van graafreductie. Er is voor combinator-graafreductie gekozen om twee redenen. Enerzijds omdat we de beschikking hadden over een aantal functionele programma's geschreven in de taal SASL die geïmplementeerd is met behulp van combinators. Anderzijds omdat deze implementatie de zwaarst denkbare belasting vormt voor het geheugenbeheer, zodat de meeste problemen in optima forma bestudeerd konden worden. Een goed begrip van de producent van garbage is van fundamenteel belang voor de behandeling van garbage collection. Naast een behandeling van een aantal aspecten van de implementatie van combinator-graafreductie presenteren we onder andere een methode om het grootste gedeelte van de cycli die tijdens reductie optreden met weinig moeite als zodanig te herkennen. Hiermee kunnen reference counting methoden zonder al te veel bezwaar worden gebruikt.

In het tweede gedeelte van het proefschrift behandelen we sequentiële graafreductie en garbage collection. Uit een statistische studie van het garbage production proces, gestuurd door in SASL geschreven applicaties, blijkt dat grafen die ontstaan tijdens het reductie proces geen uitzonderlijke eigenschappen hebben. Zo kon worden vastgesteld dat de gemiddelde diepte van de grafen evenredig is met de wortel uit het aantal knopen. Een dergelijke observatie geldt bijvoorbeeld ook voor gemiddelden over willekeurige binaire bomen. Daarna vergelijken we verschillende methoden van garbage

collection. De "mark/scan" methode blijkt het snelst, tenzij er eigenlijk te weinig geheugen aanwezig is. In dat geval is reference counting sneller dan de andere onderzochte methoden. Bij de vergelijking wordt op sub-instructie niveau de prestaties van de verschillende garbage collection methoden doorgelicht. Het blijkt dat een dergelijke gedetailleerde benadering tot dezelfde conclusies leidt als een veel gangbaarder benadering, waarbij machine instructies of zelfs "high level language statements" worden geteld. Een aspect van de interactie tussen garbage production en collection wordt gevormd door het gemiddeld aantal cellen dat per reductiestap tot garbage vervalt. We ontwikkelen een analytisch model, op grond waarvan voorspellingen omtrent dit gemiddelde kunnen worden gedaan. De voorspellingen toegepast op de implementatie van SASL blijken pessimistisch te zijn, omdat in het model vereenvoudigende aannamen zijn gemaakt. In werkelijkheid vervallen minder knopen tot garbage dan voorspeld wordt. Verfijningen van het model zijn mogelijk, maar niet onderzocht.

Het derde en laatste deel van het proefschrift behandelt een ontwerp voor de implementatie van een parallel combinator-graafreductie systeem. Onze methode is gebaseerd op de wens globale garbage collection te voorkomen. Dit houdt in, dat ten behoeve van garbage collection de verschillende verwerkingseenheden in het parallelle systeem geen interactie met elkaar mogen hebben. Binnen de parallel te verwerken berekeningen, die wij jobs noemen, vindt de normale sequentiële graafreductie en garbage collection plaats. Tussen jobs onderling is slechts beperkte samenhang mogelijk. De parallelle berekeningen vertonen een volledig hiërarchische structuur, zodat nooit meer dan één referentie naar een job aanwezig is. Het verwijderen van zo'n referentie bij de beëindiging van de job impliceert dat de bezette geheugenruimte onmiddellijk opnieuw kan worden gebruikt.

Slechts bepaalde soorten applicaties zijn geschikt om op de voorgestelde architectuur te worden uitgevoerd, zodat een belangrijke versnelling ten opzichte van sequentiële combinator graafreductie wordt bereikt. We geven richtlijnen volgens welke een functioneel programma geschikt kan worden gemaakt voor parallelle verwerking. De architectuur is ingericht op de parallelle evaluatie van verdeel-en-heers algoritmen. De richtlijnen geven de voorwaarden waaraan een programma, gebaseerd op een dergelijk algoritme, moet voldoen. De richtlijnen zijn zodanig, dat ook systemen van synchrone, communicerende processen kunnen profiteren van parallelle verwerking. We geven vijf voorbeelden van parallelle functionele programma's en berekenen de versnelling die kan worden bereikt, afhankelijk van het aantal beschikbare verwerkingseenheden. Naast de versnelling door parallelle evaluatie geven we een schatting van de benodigde hoeveelheid geheugenruimte. Deze neemt toe naarmate er meer verwerkingseenheden nodig zijn. Op grond van onze door middel van simulatie verkregen resultaten verwachten we dat een realistisch systeem met dezelfde configuraties als in de simulatie, tot vergelijkbare prestaties in staat moet zijn.

M. Amamiya, "A new parallel graph reduction model and its machine architecture," in *France-Japan artificial intelligence and computer science symp.*, pp. 329-356, North Holland, 1986.

J. L. Baer and M. Fries, "On the efficiency of some list marking algorithms," in *Information processing 77*, ed. B. Gilchrist, pp. 751-756, North Holland, Toronto, Canada, Aug. 1977.

H. P. Barendregt, *The lambda calculus, its syntax and semantics,* North Holland, Amsterdam, 1984.

H. P. Barendregt and M. van Leeuwen, "Functional programming and the language Tale," in *Current trends in concurrency: overviews and tutorials, LNCS 224*, ed. J. W. de Bakker, W.-P. de Roever, G. Rozenberg, pp. 122-207, Springer verlag, Noordwijkerhout, Netherlands, Jun. 1985.

H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeijer, and M. R. Sleep, "Term graph rewriting," in *Parle conf. on parallel architectires and languages, LNCS 259*, ed. J. W. de Bakker, A. J. Nijman, P. C. Treleaven, pp. 141-158, Eindhoven, The Netherlands, Jun. 1987.

H. P. Barendregt, M. C. J. D. van Eekelen, P. H. Hartel, L. O. Hertzberger, M. J. Plasmeijer, and W. G. Vree, "The Dutch parallel reduction machine project," *Future generations computer systems*, vol. 3, no. 4, pp. 261-270, Dec. 1987.

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

D. R. Brownbridge, "Cyclic reference counting for combinator machines," in *Second conf. on functional programming languages and computer architecture, LNCS 201*, ed. J.-P. Jouannaud, pp. 273-288, Springer verlag, Nancy, France, Sep. 1985.

N. G. de Bruijn, D. E. Knuth, and S. O. Rice, "The average height of planted plane trees," in *Graph theory and computing*, ed. R. C. Read, pp. 15-22, Academic Press, London, U.K., 1972.

T. H. Brus, M. C. J. D. van Eekelen, M. O. van Leer, and M. J. Plasmeijer, "CLEAN: A language for functional graph rewriting," in *Third conf. on functional programming languages and computer architecture, LNCS 274*, ed. G. Kahn, pp. 364-384, Springer verlag, Portland, Oregon, Sep. 1987.

F. W. Burton and M. R. Sleep, "Executing functional programs on a virtual tree of processors," in *Conf. on functional programming languages and computer architecture*, ed. Arvind, pp. 187-194, ACM, Portsmouth, New Hampshire, Oct. 1981.

F. W. Burton, "Annotations to control parallelism and reduction order in the distributed evaluation of functional programs," *ACM Transactions on programming languages and systems*, vol. 6, no. 2, pp. 159-174, Apr. 1984.

C. J. Cheney, "A non-recursive list compacting algorithm," *CACM*, vol. 13, no. 11, pp. 677-678, Nov. 1970.

D. W. Clark and S. C. Green, "An empirical study of list structure in Lisp," *CACM*, vol. 20, no. 2, pp. 78-87, Feb. 1977.

J. Cohen, "Garbage collection of linked structures," *Computing Surveys*, vol. 13, no. 3, pp. 341-367, Sep. 1981.

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

J. Fairbairn and S. Wray, "Tim: A simple lazy abstract machine to execute supercombinators," in *Third conf. on functional programming languages and computer architecture, LNCS 274*, ed. G. Kahn, pp. 34-45, Springer verlag, Portland, Oregon, Sep. 1987.

A. J. Field and P. G. Harrison, *Functional programming*, Addison Wesley, Reading, Massachusetts, 1988.

D. P. Friedman and D. S. Wise, "Reference counting can manage the circular environments of mutual recursion," *Information processing letters*, vol. 8, no. 1, pp. 41-45, Jan. 1979.

R. P. Gabriel, *Performance and evaluation of Lisp systems*, MIT Press, Cambridge, Massachusetts, 1985.

H. Glaser, C. Hankin, and D. Till, *Principles of functional programming*, Prentice Hall, Englewood Cliffs, New Jersey, 1984.

H. W. Gould and J. Kaucky, "Evaluation of a class of binomial coefficient summations," *Journal of Combinatorial theory*, vol. 1, no. 2, pp. 233-247, Sep. 1966.

R. L. Graham, "Bounds for certain multiprocessing anomalies," *Bell system technical journal*, vol. 45, no. 9, pp. 1563-1581, Nov. 1966.

P. H. Hartel, "A comparative study of three garbage collection algorithms," PRM project internal report D-23, Dept. of Comp. Sys, Univ. of Amsterdam, Feb. 1988.

P. H. Hartel and A. H. Veen, "Statistics on graph reduction of SASL programs," *Software practice and experience*, vol. 18, no. 3, pp. 239-253, Mar. 1988.

P. H. Hartel and W. G. Vree, "Parallel graph reduction for divide-and-conquer applications Part II - program performance," PRM project internal report D-20, Dept. of Comp. Sys, Univ. of Amsterdam, Apr. 1988.

P. H. Hartel, "The average size of ordered binary subgraphs," in *Graph-theoretic concepts in computer science, Int. workshop WG'88, to be published in LNCS*, ed. J. van Leeuwen, Springer Verlag, Amsterdam, Netherlands, Jun. 1988.

P. H. Hartel, *Performance analysis of storage management in combinator graph reduction*, Dept. of Comp. Sys, Univ. of Amsterdam, Oct. 1988. PhD. Thesis

P. H. Hartel, "Performance of compiled fixed combinator graph reduction," PRM project internal report D-27, Dept. of Comp. Sys, Univ. of Amsterdam, Aug. 1988.

T. Hikita, "On the average size of Turner's translation to combinator programs," *Journal of Information Processing*, vol. 7, no. 3, pp. 164-169, 1984.

C. A. R. Hoare, "Optimization of store use for garbage collection," *Information processing letters*, vol. 2, no. 1, pp. 165-166, 1974.

R. F. H. Hofman, "An on-the-fly scheduling algorithm for an experimental parallel reduction machine," PRM project internal report D-18, Dept. of Comp. Sys, Univ. of Amsterdam, May. 1988.

P. Hudak and B. Goldberg, "Serial combinators: "Optimal" grains of parallelism," in *Second conf. on functional programming languages and computer architecture, LNCS 201*, ed. J.-P. Jouannaud, pp. 382-399, Springer verlag, Nancy, France, Sep. 1985.

P. Hudak and B. Goldberg, "Distributed execution of functional programs using Serial combinators," *IEEE Transactions on computers*, vol. C-34, no. 10, pp. 881-891, Oct. 1985.

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

T. Johnsson, "Efficient compilation of lazy evaluation," *Sigplan Notices*, vol. 19, no. 6, pp. 58-69, Jun. 1984.

T. Johnsson, "Lambda lifting: transforming programs to recursive equations," in *Second conf. on functional programming languages and computer architecture, LNCS 201*, ed. J.-P. Jouannaud, pp. 190-203, Springer verlag, Nancy, France, Sep. 1985.

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

R. M. Keller and F. C. H. Lin, "Simulated performance of a reduction based multiprocessor," *IEEE computer*, vol. 17, no. 7, pp. 70-82, Jul. 1984.

R. M. Keller, "Distributed graph reduction from first principles," Technical report, Dept. of Comp. Sci, Univ. of Utah, 1985.

R. B. Kieburtz, "The G-machine: A fast, graph-reduction evaluator," in *Second conf. on functional programming languages and computer architecture, LNCS 201*, ed. J.-P. Jouannaud, pp. 400-413, Springer verlag, Nancy, France, Sep. 1985.

W. E. Kluge, "Cooperating reduction machines," *IEEE Transactions on computers*, vol. C-32, no. 11, pp. 1002-1012, Nov. 1983.

D. E. Knuth, *The art of computer programming, volume 1: Fundamental algorithms*, Addison Wesley, Reading, Massachusetts, 1973. second edition

D. E. Knuth, *The art of computer programming, volume 2: Seminumerical algorithms*, Addison Wesley, Reading, Massachusetts, 1980. second edition

P. W. M. Koopman, "Interactive programs in a functional language: A functional implementation of an editor," *Software practice and experience*, vol. 17, no. 9, pp. 609-622, Sep. 1987.

P. J. Landin, "The mechanical evaluation of expressions," *Computer Journal*, vol. 6, no. 4, pp. 308-320, Jan. 1964.

G. A. Magó, "A network of microprocessors to execute reduction languages - Part I," *International journal of computer and information sciences*, vol. 8, no. 5, pp. 349-385, Oct. 1979.

G. A. Magó, "A network of microprocessors to execute reduction languages - Part II," *International journal of computer and information sciences*, vol. 8, no. 6, pp. 435-471, Dec. 1979.

D. L. McBurney and M. R. Sleep, "Transputer based experiments with the ZAPP architecture," in *Parle conf. on parallel architectires and languages, LNCS 259*, ed. J. W. de Bakker, A. J. Nijman, P. C. Treleaven, pp. 242-259, Eindhoven, The Netherlands, Jun. 1987.

P. H. Michielse and H. A. van der Vorst, "Data transport in Wang's partition method," Internal report 86-32, Dept. of Comp. Sci, Technical Univ. Delft, 1986.

J. C. Mulder, "Complexity of combinatory code," Technical report nr 389, Dept. of Mathematics, Univ. of Utrecht, Aug. 1985.

Motorola Inc., *MC68000 8-/16-/32-Bit microprocessors programmer's reference manual,* Prentice hall, Englewood Cliffs, New Jersey, 1986. fifth edition

N. J. Nilsson, *Problem solving methods in artificial intelligence,* McGraw-Hill, 1971.

S. L. Peyton Jones, "An investigation of the relative efficiency of combinators and lambda expressions," in *ACM symp. on Lisp and functional programming*, pp. 150-158, ACM, Pittsburg, Pennsylvania, Aug. 1982.

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

S. L. Peyton Jones, "Yacc in SASL - an exercise in functional programming," *Software practice and experience*, vol. 15, no. 8, pp. 807-820, Aug. 1985.

S. L. Peyton Jones, "Using Futurebus in a fifth-generation computer," *Microprocessors and microsystems*, vol. 10, no. 2, pp. 69-76, Mar. 1986.

S. L. Peyton Jones, *The implementation of functional programming languages,* Prentice Hall, Englewood Cliffs, New Jersey, 1987.

C. A. Ruggiero and J. Sargeant, "Control of parallelism in the Manchester dataflow machine," in *Third conf. on functional programming languages and computer architecture, LNCS 274*, ed. G. Kahn, pp. 1-15, Springer verlag, Portland, Oregon, Sep. 1987.

M. Schönfinkel, "Über die bausteine der mathematischen logik," *Mathematische Annalen*, vol. 92, no. 6, pp. 305-316, 1924.

W. R. Stoye, T. J. W. Clarke, and A. C. Norman, "Some practical methods for rapid combinator reduction," in *ACM symp. on Lisp and functional programming*, pp. 159-166, ACM, Austin, Texas, Aug. 1984.

W. R. Stoye, *The implementation of functional programming languages using custom hardware,* Univ. of Cambridge, U.K., Dec. 1985. PhD. Thesis

P. C. Treleaven and R. P. Hopkins, "A recursive computer for VLSI," *9-th IEEE/ACM symp. on computer architecture, Computer architecture news*, vol. 10, no. 3, pp. 229-238, Austin, Texas, Apr. 1982.

D. A. Turner, "A new implementation technique for applicative languages," *Software Practice and Experience*, vol. 9, no. 1, pp. 31-49, Jan. 1979.

D. A. Turner, "Another algorithm for bracket abstraction," *Journal of symbolic logic*, vol. 44, no. 2, pp. 267-270, Jun. 1979.

D. A. Turner, "SASL language manual," Technical report, Computing Laboratory, Univ. of Kent at Canterbury, Aug. 1979.

D. A. Turner, "The semantic elegance of applicative languages," in *Conf. on Functional programming languages and computer architecture*, ed. Arvind, pp. 85-92, ACM, Portsmouth, New Hampshire, Oct. 1981.

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

F. Tuynman and L. O. Hertzberger, "A distributed real-time operating system," *Software practice and experience*, vol. 16, no. 5, pp. 425-441, May. 1986.

D. Ungar, "Generation scavenging: a non-disruptive high performance storage reclamation algorithm," *Software engineering symp. on practical software development environments, SIGPLAN Notices*, vol. 19, no. 5, pp. 157-167, ACM, Pittsburg, Pennsylvania, Apr. 1984.

A. H. Veen, "Analytic modelling of a parallel graph reducer," Technical Report 79, Dept. of Comp. Sci, Univ. of Nijmegen, May. 1986.

W. G. Vree, "The grain size of parallel computations in a functional program," in *Conf. on Parallel processing and Applications*, ed. E. Chiricozzi, A. d'Amico, pp. 363-370, Elsevier Science Publishing, L'Aquila, Italy, Sep. 1987.

W. G. Vree, "A parallel hydraulical simulation program in SASL," PRM project internal report D-13, Dept. of Comp. Sys, Univ. of Amsterdam, Oct. 1987.

W. G. Vree, "On the need of cycles in graph reduction," PRM project Internal report D-16, Dept. of Comp. Sys, Univ. of Amsterdam, Nov. 1987.

W. G. Vree and P. H. Hartel, "Parallel graph reduction for divide-and-conquer applications Part I - program transformation," PRM project internal report D-15, Dept. of Comp. Sys, Univ. of Amsterdam, Apr. 1988.

W. G. Vree, ''Parallel graph reduction for communicating sequential processes,'' PRM project internal report D-26, Dept. of Comp. Sys, Univ. of Amsterdam, Aug. 1988.

P. Wadler, ''Listlessness is better than laziness,'' in *ACM symp. on Lisp and functional programming*, pp. 45-52, ACM, Austin, Texas, Aug. 1984.

C. P. Wadsworth, *Semantics and pragmatics of the lambda calculus,* Oxford Univ, U.K., 1971. PhD. Thesis

H. H. Wang, ''A parallel method for tri-diagonal equations,'' *ACM transactions on Mathematical Software*, vol. 7, no. 2, pp. 170-183, Jun. 1981.

P. Watson and I. Watson, ''Evaluation of functional programs on the Flagship machine,'' in *Third conf. on functional programming languages and computer architecture, LNCS 274*, ed. G. Kahn, pp. 80-97, Springer verlag, Portland, Oregon, Sep. 1987.

R. P. Weicker, ''Dhrystone: A synthetic systems programming benchmark,'' *CACM*, vol. 27, no. 10, pp. 1013-1030, Oct. 1984.

N. Wirth, *Programming in Modula-2,* Springer verlag, Berlin, 1982. second edition