The Implementation of Functional Languages on Parallel Machines with Distributed Memory

Marco Kesseler

The Implementation of Functional Languages on Parallel Machines with Distributed Memory Marco Kesseler 1996

The Implementation of Functional Languages on Parallel Machines with Distributed Memory

een wetenschappelijke proeve op het gebied van de
Wiskunde en Informatica

Proefschrift

ter verkrijging van de graad van doctor
aan de Katholieke Universiteit van Nijmegen,
volgens besluit van het College van Decanen in het
openbaar te verdedigen op maandag 29 april 1996
des namiddags om 1.30 uur precies

door

Marcus Henricus Gerardus Kesseler

geboren op 10 april 1968 te Heumen

Promotor:	prof. dr. ir. M.J. Plasmeijer
Co-promotor:	dr. M.C.J.D. van Eekelen

Manuscriptcommissie:

prof. dr. R. Loogen	Philipps-Universität Marburg
dr. H. Kuchen	Rheinisch Westfälische Technische Hochschule Aachen
dr. P. Hartel	Universiteit van Amsterdam & University of Southampton

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Kesseler, Marcus Henricus Gerardus

The implementation of functional languages on parallel machines with distributed memory / Marcus Henricus Gerardus Kesseler. - [S.l. : s.n.]. - Ill.
Proefschrift Katholieke Universiteit Nijmegen. - Met Index, lit. opg. - Met samenvatting in het Nederlands.
ISBN 90-9009247-1
Trefw.: implementatie / functionele programmeertalen / parallelisme

Uproot your questions from their ground and the dangling roots will be seen. More questions!

Frank Herbert, Chapter House Dune

Contents

Acknowledgements

These very first words have actually been written the last. I have mixed feelings about thanking specific people. I would rather not have a little list of names with noisy empty spaces. But who is going to read a lengthy list? So, I hereby express my appreciation to anyone who inspired me.

Still, it takes some carefully chosen silences to make some music and some people suffer the most. John van Groningen and Ronny Wichers Scheur have had to share the same room with me for the last couple of years. In particular, I thank John for not being susceptible to any form of hype, and for his skilfully designed software. Ronny I thank for keeping me sharp - he likes juggling - and for sharing valuable insights in the art of designing graphical user interfaces. Skills and arts combined in a single room.

Then, there exist some people who actually wanted to be part of the manuscript committee and read through the whole lot of this thesis. They will tell you it is their job. Well, do not be fooled, they do have a choice. Rita, Herbert and Pieter, I thank you for all the hours I have stolen from you.

Rinus, you allowed me to find my own ways. You also carefully prepared me for military service by giving me two of your cats. They certainly find their ways! Together with Marko van Eekelen and Henk Barendregt you have created a place far away from mind-boggling bureaucracies. I do not have to convince you that procedures and creativity do not mix well. Luckily, we have people like Mariëlle van der Zandt and Jacqueline Parijs to deal with all the messy imperative details.

And finally, I thank the people that are dearest to me. My parents have always supported me, but not without warning me that paper stuff has its limits. My sister Sandra knew that all along. I thank her for showing me that I am not always right. More than I can possibly say, I thank Monique for her support, patience and understanding during the many hours that I had my mind set on writing.

1. Introduction

This thesis explores the implementation of functional languages on parallel hardware with distributed memory. In this chapter we will determine the most interesting issues, followed by a general outline of our research.

We will start with some observations on the limits of software development and hardware development. The former is very complex, which puts a strain on programmers, and the latter has its physical limits, which puts a bound on efficiency. Functional programming languages have advantages in both areas. They allow the construction of concise programs that are suited for evaluation on parallel hardware. The main disadvantage of functional programming languages is that they are difficult to implement efficiently, in particular on parallel machines. In this thesis, we will investigate these implementation problems, by building and evaluating a parallel implementation of the functional language *Concurrent Clean* on concrete parallel *transputer* hardware.

Concretely, the structure of this chapter is as follows. Section 1.1 will focus on the limits of software development and hardware development. In section 1.2 we will elucidate the advantages of functional programming. Section 1.3 will show the major issues for obtaining an efficient implementation of a functional language (in particular on parallel hardware). Section 1.4 will give a general outline of our research. Section 1.5 and 1.6 will explain the characteristics of Concurrent Clean and the transputer processor. And finally, the last section will give an overview of this thesis.

1.1. The limits of software development and hardware development

The development of correct software is known as a hard problem. Often, the construction and maintenance of large and complex systems imposes serious problems. In particular parallel systems are notoriously difficult to program. Much research is devoted to solving these problems and many different directions can be taken. This thesis focuses on the development of new - and more powerful - programming languages, more precisely it investigates *functional programming languages*. It will limit itself to one aspect, namely that of programming *parallel* machines using a functional language. Before we proceed, we will take a closer look at the problems found in software development.

1.1.1. The limits of nature

Realising that faster computers lead to a better productivity, hardware manufacturers have been urged to devise ever faster processors. Over the years, computer hardware has become increasingly powerful. So far, processor speed has doubled every few years. And although some have claimed that this development inevitably will stop at some point in the future, there is no sign that this moment will arrive shortly.

In spite of this, processor manufacturers will have to deal with a fundamental speed limit: the velocity of light. More and more, not the speed of an operation will be the limiting factor, but the time it takes to fetch the arguments. At best, these travel with the speed of light. Apart from this, current technology puts a boundary on the speed at which switches can operate. In this area there might turn out to be some hard boundary as well. These limits restrain the possibility of using of ever higher clock frequencies to obtain higher processor speed.

To deal with these issues (to some extent), processors are emerging that use pipelines of several simple functional units on a single chip. The simplicity of each unit allows it to operate at relatively high frequencies, while enough functional power is maintained by overlapping the operation of several units. For instance, the arguments of some subsequent arithmetic operation might be fetched while another arithmethic operation is in progress. In this way the delays that are involved in obtaining the arguments can be hidden.

Similarly, some processors contain independent parallel units for different sorts of operations, so that various tasks can be achieved simultaneously, superseding the functional ability of traditional designs. Such a *super-scalar* processor might perform several operations in a single clock-tick. These designs also permit instructions to be executed out of order, which allows critical data paths to become shorter. In the near future, similar techniques will become increasingly important for achieving the necessary increase in processor speed.

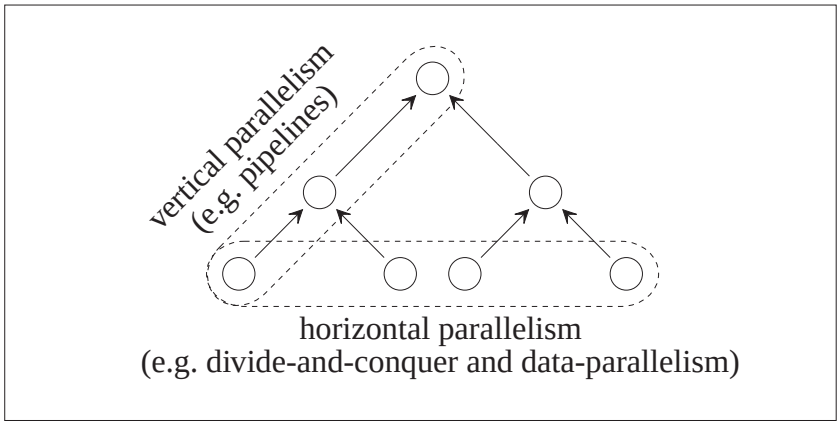


Figure 1-1: *Types of Parallelism.*

The main drawback of such solutions, is that they assume that a certain level of parallelism is inherently present in *sequential* programs and that it is easily detectable. So far, the required level is quite low and many sequential programs indeed contain this quantity. It can be transferred to the machine code, provided that compilers *schedule*

instructions in such a way that dependencies between adjacent instructions are avoided. One may question the viability of these methods when greater processing speeds are to be achieved. The inherent parallelism in sequential programs may be insufficient to keep the processor busy. But what is more, a compiler may not even be able to extract enough information to exploit the parallelism that *is* available. If greater speeds are to be obtained, one needs to develop explicitly parallel programs.

These parallel programs on the other hand, may easily exhibit more parallelism than a single processor chip is able to handle. A single chip can contain only a limited number of functional units. If we want to exploit more parallelism, so that programs run at greater speed, we need to construct a machine with multiple chips, each containing one or more functional units. Speed is not the only reason for constructing such *parallel architectures*. In contrast to purely sequential architectures, parallel ones commonly contain large numbers of identical parts. This can seriously decrease hardware development costs. In short, we need a *parallel architecture* that contains a reasonable number of (identical) functional units.

Concurrently executing functional units appear to be the key feature of a parallel architecture. But, at a low level, sequential architectures have always incorporated some form of parallelism. For example, the addition of integers is often implemented by a number of parallel operations. So, multi-chip parallel machines do not seem fundamentally different from architectures with only a single 'sequential' processor. However, the tasks that a parallel machine can perform concurrently should be fairly complex, i.e. they should at least have the complexity of a single assembly level machine instruction. Furthermore, machines with only a few parallel functional units can hardly be regarded as parallel machines of much importance. Therefore, we will only consider parallel machines that contain a fair number of processors (at least 10, preferably more than 100) and that are suited to be extended to larger numbers.

A number of parallel machines have been constructed in the (recent) past. Each with its own characteristics, matching the purpose for which it was built. Only few of these architectures have been truly successful, mainly in areas where the actual problem turned out to be parallelisable in a relatively easy way (in particular vector machines exploiting data-parallelism). There are several reasons for the failure of others, but one of the most significant ones, is that in general, parallel machines are harder to program than sequential ones. In theory, each of the newly devised parallel machines was far more powerful than their sequential counterparts, but in practice it proved to be extremely difficult to program it in such a way that this power became available. It some areas it has turned out to be more cost-effective to simply wait for a faster processor, than to spend a lot of resources - and time - on the development of (complex) parallel algorithms.

So, it has become apparent that building machines with many parallel functional units does not solve all problems. A great challenge lies in programming them. We will take a closer look at this below.

1.1.2. The limits of man: the software crisis

Programmers face an increasing complexity in the problems they have to solve. As the speed of sequential computation increases, more complex problems can be handled. Consequently, software becomes ever more complex, or at least it should. Moreover, in some areas, parallelism needs to be exploited in order to surpass the speed of a single processor. This introduces a considerable amount additional complexity.

However, developing complex software is hard and often, software products are unreliable, unmanageable and unprovable. Hardly any software system exists that does not contain bugs. This situation is known as the software crisis. To deal with this problem, we need more powerful programming languages, methods, and tools. Over the years these have indeed improved, but not sufficiently to overcome the software crisis. In contrast, as the importance of computer systems has grown and their complexity has increased, the problems have got worse.

Consequently, much research is devoted to solving the following two questions:

- How can we correctly develop large software systems at low cost?

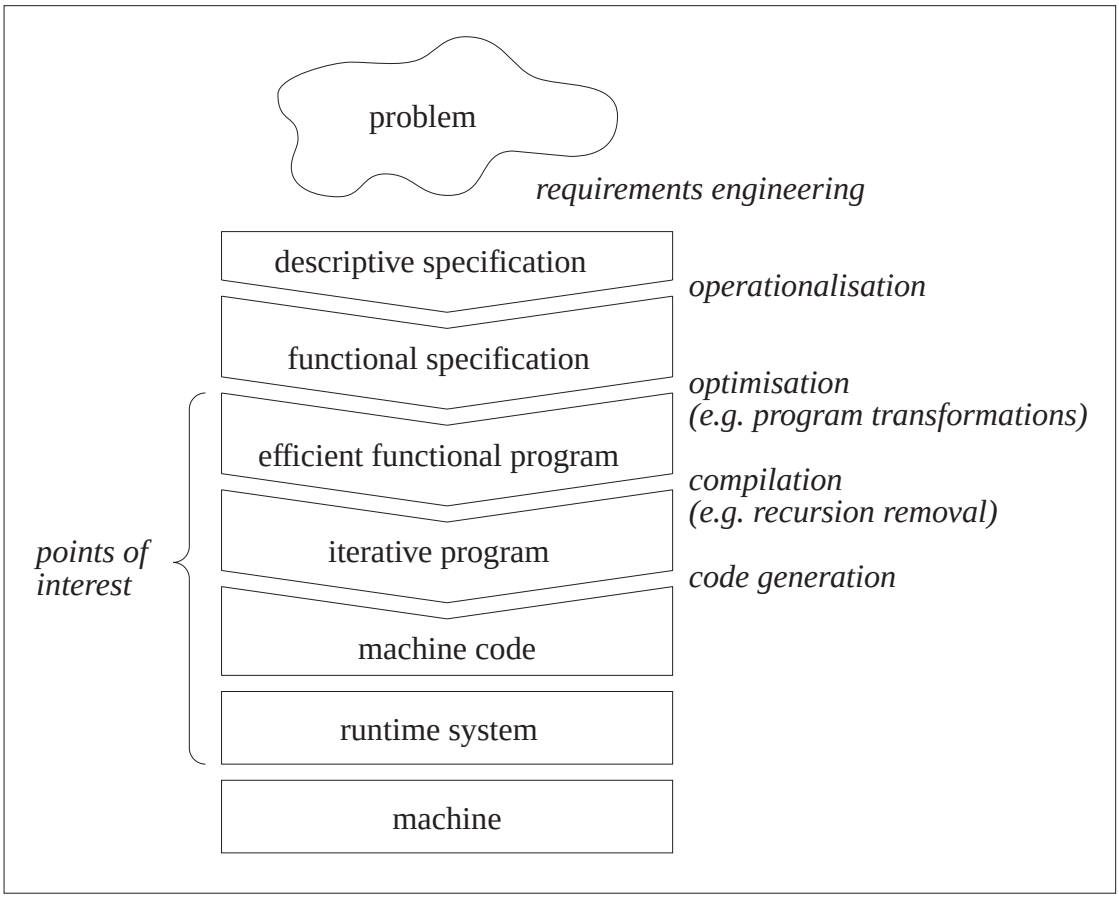


Figure 1-2: *The position of research on functional programming languages within the software development track.*

- How can we increase processing speed at low cost? In particular, how can we correctly develop large parallel software systems at low cost?

Many different approaches have been taken to achieve these goals, many of which are complementary. Examples are the investigation of software engineering techniques, and research on program transformation techniques. We have chosen to focus on the development of new programming languages, specifically, the development of functional programming languages. Figure 1-2 clarifies which part of the software development traject our research addresses.

1.2. The promise of functional programming

The basic - and only - building block of functional languages are functions. Expressions can only be constructed by means of function application. As in mathematics, a function maps objects of a domain to objects of a range. Most elementary however, is that a function can map its argument objects to at most one object in its range. This object is then called *the* image of the object in the domain. This implies that the function result is entirely defined by its arguments.

Pure functional languages will strictly maintain this property[1], and this has an important consequence: compared to traditional imperative programming languages, pure functional ones have the advantage of ensuring *referential transparency*. A certain (sub)expression always means the same (provided it has a meaning). This is certainly not the case in imperative languages. A well-known example is the assignment statement which imperative languages commonly support. A variable *v* may be assigned various values over time, so that the expression *v* gets a different meaning each time another value is assigned to it. Clearly, functional languages cannot allow assignments, and consequently they do not have any variables. In contrast, every object has a constant meaning. This has important advantages.

First of all, the term ‘referential transparency’ is merely indicating that objects in a functional programming language have the same characteristics as mathematical functions. Having such functions, we can prove their properties by means of ordinary mathematical methods (equational reasoning). No additional proof techniques are necessary. This also implies that it very easy to perform (semi-) automatic program transformations on functional programs.

Secondly, if referential transparency is ensured it also becomes easier to reason about programs in more intuitive ways. To derive the meaning of a statement in a functional language one only has to consider the expression itself, and not various other conditions, such as the value of global variables. Sentences mean the same in any context. One can reason about a functional program by splitting it up in parts and then consider these parts

---

1 Examples of pure functional languages are Miranda (Turner, 1985), LML (Augustsson, 1984), Haskell (Hudak et al., 1992) and Concurrent Clean (Plasmeijer and van Eekelen, 1993). LISP (McCarthy, 1960) and ML (Harper et al., 1986) are examples of impure functional languages, which sometimes allow a function result to depend on other factors than merely its arguments. One might question whether the adjective ‘functional’ is appropriate for these languages.

independently of each other. A functional language provides modularity and clear interfaces in itself.

And finally, referential transparency makes it possible to change the order in which expressions are evaluated without altering the meaning of the program. Each statement always means the same, so it does not matter when it is evaluated. This has serious advantages. On the one hand, this property makes it possible to use *lazy evaluation*, which means that functions are only evaluated at the moment they are needed, if they are needed at all (call-by-need). Apart from avoiding superfluous work, this has the advantage that infinite structures can be dealt with in a very elegant way, leading to a clear way of programming. On the other hand, the possibility to safely change the evaluation order makes it possible to easily introduce parallelism. If a function has multiple arguments, they all can be evaluated in parallel without changing the outcome of the function. As a result, functional languages are inherently parallel, in contrast to imperative ones, which depend on the execution of correctly ordered sequences of instructions. This means that functional programming is a very promising candidate for solving the problem of programming complex parallel systems.

In other words, functional programs reduce the over-specification - to a large extent the unnecessary sequentialisation - that is commonly part of solutions written in imperative languages. Functional programs are less explicit on *how* an answer should be computed, implying that programmers *can* be less explicit in this respect and concentrate more on the essence of programming a solution. They only have to specify *what* the solution should be. The downside of all this, is that implementing a functional language is relatively complex compared to implementing an imperative one. A functional language implementation has to derive the exact way a program should execute, whereas imperative programs specify this explicitly. We will have a closer look at this in the next section.

1.3. The problem of implementing functional languages

As we have seen above, functional languages have important advantages. Nonetheless, they have hardly been used by software developers. To explain this, we need to consider the following aspects of programming:

- Functional languages sometimes have difficulties with interfacing to traditional hardware and software.
- The use of functional languages suffers from the common imperative background of programmers. Programmers need to be trained to use functional languages.
- There is a lack of a powerful programming environments for functional languages.
- There has been a serious shortage of efficient implementations of functional languages for different computer architectures, in particular for parallel hardware.

A few years ago, functional programs executed at least an order of a magnitude slower than their imperative counterparts. As a result, the advantages of functional programming were rather hypothetical. This situation has been improving. Sequential implementations have become available that can be used to develop serious programs. But still, there are many problems that need to be solved. In particular, the implementation

techniques for parallel functional languages have not yet matured. In contrast to sequential implementations, hardly any usable general purpose parallel implementation exists of a pure functional language[2]. This not surprising if we consider the more complex nature of parallel systems. Therefore, this thesis will mainly focus on implementation techniques for parallel architectures.

The following subsections will identify the most important implementation problems. This will make clear that some efficiency problems exist that cannot be solved automatically. Consequently, the programmer must deal with these. We need to find ways to control efficient (parallel) programming explicitly. Amongst others, this implies that we should have a close look at the runtime behaviour of parallel programs.

1.3.1. Sequential implementation issues

Even though we are focusing on the realisation of parallel implementations, we cannot completely ignore sequential implementation techniques, and in particular sequential code quality. First of all, one should take into account the significant advances that have been made in sequential implementation techniques for functional languages. If it is possible to use similar techniques for parallel architectures, this would greatly reduce the amount of work. And secondly, parallel machines usually consist of a number of sequential processors that run concurrently. In the past, parallel architectures typically contained custom sequential processors that incorporated additional support for communications, such as the transputer hardware and the CM-2 connection machine. Since then it has become common practice to construct parallel machines with standard sequential general purpose processors, interconnected by some high performance network. For instance, the CM-5 connection machine consists of ordinary Sparc processors that are interconnected by a custom tree-like network. Likewise, some manufacturers have constructed parallel machines with modern RISC processors, such as the PowerPC, using transputers [sic!] merely to perform the communications. In either case, the performance of a parallel implementation will greatly depend on sequential code quality.

Therefore, we need to know if both sequential and parallel implementation techniques can be joined into one efficient system. Parallel implementation techniques should not severely restrict the use of significant sequential techniques and vice versa. We need not however, concentrate on the development of sequential techniques. Many have already focused on this topic. Instead, we will investigate how parallel implementation techniques can be combined with existing sequential ones. To do this thoroughly, we need to understand the problems that are related to sequential implementations, and the nature of their solutions.

One of the most interesting aspects in this respect is dealing with *state* when referential transparency has to be maintained. An important example of a system with state is the real world. If some program performs I/O it actually changes the state of the world. Functional languages find this problematic because an object with some state is just a variable. Changing its state means that another value has to be assigned to it. Functional

---

[2] Although special purpose languages like SISAL and Erlang have had some success in very specific area's. Erlang has been used to program parts of a telephone system, whereas SISAL supports efficient vector-like processing. Nonetehless, these languages are not widely used.

languages cannot do this. Instead, they need to create a *new* object containing the altered state. This is problematic if the object is large, or if it represents a physical object in the real world. In contrast, imperative languages can simply update the old object with its new value. They are able to change objects. Often only a small part of an object needs to be altered, which is much more efficient than creating a entirely new object.

Many of these efficiency problems can be avoided if more sophisticated implementation technologies are developed. For instance, the research on *uniqueness typing* has made it possible in certain cases to update objects in place on the machine level, while on the language level it seems that new objects are created. Amongst others. this enabled the development of a sophisticated I/O system (Achten and Plasmeijer, 1995). In this thesis, we will demonstrate that uniqueness typing is very important for parallel implementations as well and we will show how this technique can be incorporated in a parallel system. Not only does this allow the incorporation of a parallel I/O system, but for a number of programs this also formed the basis of considerable parallel speed-ups.

With respect to speed deficiencies in sequential code, we do not claim that all can be cured, but they may very well become acceptable. The gain in ease of programming will then outweigh some loss of efficiency. In a similar way we accept some loss of efficiency by using high-level imperative languages instead of assembly languages. This consideration is important for parallel machines as well, although some might argue that execution speed is far more important than ease of programming for such machines. However, it may be more cost-effective to simply add some extra processors, than to develop a slightly faster parallel program in an imperative language.

1.3.2. Parallel implementation issues

For parallel implementations sequential implementation techniques are important, but one faces a number of additional problems that need to be solved if an efficient parallel implementation is to be obtained. Some of these are of a quite technical nature. For instance, one has to manage processes and - if one uses a machine with distributed memory- one has to provide reliable communications. This often comes down to picking a (known) solution and testing it. This is not as trivial as it might seem. The solutions are often complex, and some have hardly been tested in reality, or at best on different architectures.

In contrast to these problems, there exist others that are of a more fundamental nature. For instance, memory management - and in particular garbage collection -, is not only rather hard to realise on a parallel machine (in particular with distributed memory), but in addition, the problem has not yet been completely solved. For example, it is not yet known in what way distributed cyclic structures can be removed efficiently from machines with distributed memory. Some solutions have been suggested, but these hardly seem to be of any practical use. Other problems become apparent when considering load balancing techniques. A number of compile-time and runtime techniques have been developed, but these only perform well if certain algorithmic conditions are met. In particular, no general method is known that will automatically split up a computation in several parallel sub-computations in a satisfactory way. We will take a closer look at this in the following section.

1.3.3. Implicit versus explicit parallelism

It is not yet clear what is the best way to introduce parallelism in functional programs. This may sound strange, as functional programs are inherently parallel, but the point is that they actually contain too much parallelism. If one tries to exploit it all, a lot of overheads will be introduced. So the question is not so much how to introduce parallelism, but rather how to throttle it in a sensible way.

Unfortunately, there is no known technique that will automatically derive the best way to split up a program in parallel parts under all circumstances. Basically, it is not (yet) possible to get accurate information on all relevant aspects in a parallel system. First of all, one would require details of the implementation itself, and this has not yet been established. Secondly, a serious problem is formed by a fundamental lack of information on the complexity of computations. This information usually depends on runtime conditions. As a result, compile-time analysis is not accurate enough, nor are runtime facilities if future conditions have to be taken into account. A runtime mechanism additionally, has the disadvantage that it may need to gather information from various processors in a distributed system. The required communications not only introduce runtime overheads, but they also introduce the risk of obtaining outdated information.

Dealing with these problems presents an extraordinary amount of work in addition to the problems mentioned earlier. It is even questionable whether it is possible at all to find a mechanism that automatically introduces parallelism properly. Therefore, we will avoid this problem entirely.

Instead of introducing parallelism implicitly, we will allow the programmer to indicate it explicitly by means of special constructs. Choosing this approach, we burden the programmer with extra work. To limit the drawbacks of this method the constructs for specifying parallelism must be as clear as possible, without requiring too detailed knowledge of the actual language implementation. Clearly, one should take care not to sacrifice the benefits of high level programming. Furthermore is becomes necessary to devise tools for helping the programmer. One could think of semi-automatic transformational derivation of parallel programs. Ideally, such a system should make it possible to start with a functional specification and guide the programmer to an efficient program that contains explicit constructs for parallelism.

To some extent, we will examine the runtime semantics of explicit constructs for parallelism. We will have a close look at some primitives for starting up parallel processes and we will see how we can use these to construct *skeletons* that enable the easy use of certain parallel programming paradigms. The design of special tools falls outside the scope of this thesis. Such tools become useful at the moment that it is clear what forms of parallelism can effectively be exploited in a functional language, but unfortunately, this is not yet entirely the case.

1.3.4. The runtime behaviour of functional programs

The problems depicted above, are not restricted to parallel implementations only. In general, it is impossible for a compiler to derive the most efficient code regardless of the way a solution has been specified. Consequently, two different functional programs that are equivalent from a formal point of view (delivering the same result), might behave rather

differently (with respect to the required time and space). A programmer cannot ignore this. He will need to explicitly program the most efficient solution.

But, as we have seen above, the standard constructs of a functional language might not be sufficient to accomplish this. Even if a solution has been programmed in the most efficient way, the implementation may still lack some information that is necessary to automatically derive the best code. The compiler will have to make assumptions and these influence code quality. Consequently, the same functional program may exhibit quite different operational behaviour, depending on the actual implementation of the language. To avoid such problems, and to increase code quality, one must have additional ways to explicitly control the efficiency of functional programs.

This implies that one should be able to reason about the runtime behaviour of functional programs. Unfortunately, the abstract nature of functional languages makes this rather hard. First of all, functional languages tend to obscure their operational behaviour, simply because a number of problems are handled automatically by the implementation. The implicit handling of memory allocation for instance, makes it difficult to understand the memory demands of certain functions. Likewise - as will become clear later -, communications may take place implicitly in a parallel implementation. A programmer may have trouble to extract the size and kind of the information that is transmitted, or even to determine whether any communication takes place at all.

Additionally, operational behaviour becomes less clear, because of the freedom in the order of evaluation that functional languages offer. If lazy evaluation is employed it becomes more difficult to reason about the time - and space - it will take to evaluate a function. One is forced to take into account the evaluation of previously unevaluated arguments. It is already hard to figure out which these are, let alone to determine the costs of evaluating them. Furthermore, if an implementation employs strictness analysis to introduce eager evaluation at some points this can have serious effects on performance, and thus, on runtime behaviour. In general, as the implementation decides which evaluation strategy it uses, it becomes more difficult for a programmer to derive operational behaviour.

These problems already exist in sequential implementations, but they are worse in parallel ones, because one has to split up a problem into parallel tasks. The uncertainties with respect to operational behaviour can make it extremely hard to design efficient parallel programs. For this reason we will not only consider implementation techniques, but we will also reflect on some aspects of designing efficient parallel algorithms in a functional language. We will need to determine the suitability of functional programming languages to express parallel algorithms. In particular, we need to investigate the expressiveness of parallel constructs.

1.4. An outline of our research

From the above, it follows that we should basically be focusing on implementation issues with respect to explicitly parallel constructs in a functional language. In this section we will give a short overview of questions we will address, and the way we intend to answer them.

1.4.1. The research questions

Two questions need to be answered. First of all, we need to investigate the expressive power of the parallel constructs. And secondly, we need to know if - and how - they can be implemented effectively.

1. The expressive power of parallel constructs depends on a number of factors.
   a) Firstly, one needs to establish the expressive power of a set of constructs as a group. Are they suited to describe a large set of parallel systems, or only a small one? In other words, do they provide a general purpose parallel language or only a very specialised one?
   b) Secondly, one needs to determine the expressive power of each construct on its own. This influences the number of constructs we need to solve a certain problem, and therefore the conciseness of the solution.
   c) And finally, we need to know if parallel constructs are comprehensible. Is it easy to reason about their behaviour? This will turn out to be one of the hardest questions, in contrast to the ones above, which are rather trivial.
2. Still, in a programming language, there is no use for very expressive constructs if they cannot be implemented effectively. We need to consider two aspects.
   a) On the one hand, we need to know *how* a construct can be implemented most efficiently on a particular architecture.
   b) Once we have established a suitable way to implement a certain construct, we should determine if the construct behaves as intended in reality. The relation between theory and practice must become apparent. It will be clear, that in order to reason about programs, there must be no serious discrepancy. This is closely related to question *1a*.

A serious language must be able to describe real solutions. This means that the questions above not only must be answered for small systems, but also for non-trivial large ones. Amongst others, this puts serious demands on execution speed, memory management, process management and communication mechanisms. This will be reflected in the implementation techniques we will consider.

At the same time, we will make no concessions with respect to the generality of the implementation. We will not ignore certain forms of parallelism for the sole reason that they complicate implementation. We will not limit ourselves to *divide-and-conquer* style parallelism, and in particular we will not exclude *speculative parallelism.* This sharply contrasts with other research projects in this field, such as HyperM (Barendregt *et al*., 1992), and the ZAPP implementation of Concurrent Clean (Goldsmith *et al,* 1993).

1.4.2. The research plan

The approach we have taken to answer the questions above, is to realise a state-of-the-art parallel implementation and to test it. This is needed to gain insights in implementation techniques and to evaluate theories about expected parallel behaviour. Simulations and theoretical models may not be accurate enough to supply this information.

To accomplish this we will use an iterative procedure that takes a number of steps. First we will realise the implementation of some parallel constructs that we deem most useful. Next, we will develop and run test programs, in order to evaluate the constructs and their implementation. This will give us some new insights, which we can use to improve the implementation of current constructs, or to develop new ones. The latter can either be 'stand-alone' constructs, or built on top of old ones. This will take us back to the first step of this paragraph.

1.4.3. The architecture

One cannot possibly develop an implementation for all existing - or future - architectures simultaneously. For practical reasons we need to limit our focus. Therefore we have restricted the extent of our research in the following ways.

First of all, we have chosen not to develop a special purpose machine for executing functional programs. Instead we concentrate on realising an implementation for standard general purpose machines only. On the one hand, special purpose machines are relatively expensive, and they are not used by many people (the former implies the latter and vice versa). This obstructs our goal to get functional programming generally accepted by a large community. On the other hand, it takes more effort to maintain a system that is based on specialised hardware, than to accomplish the same using general purpose machines. Not only does one need to keep up with the latest implementation techniques for functional languages, but additionally, one needs to adapt the architecture to state-of-the-art hardware construction techniques. The latter is rather hard, considering the massive efforts that the major industries continue to put in the development of traditional von Neumann architectures. These have become tremendously powerful at such a rate that they can easily take over tasks of more specialised architectures. The end of this development is not yet in sight. Consequently, using special purpose hardware does not always guarantee that programs run at the highest possible speed. Indeed, the results of experiments with implementations of functional languages on specialised hardware have been rather disappointing so far, and this direction of research has largely been discontinued (Darlington and Reeve, 1981; Watson and Watson, 1987; Peyton Jones *et al*., 1987; Hankin *et al*., 1985; Richards, 1985; Stoye, 1985; Keller *et al*., 1984; Anderson *et al*., 1987; Magó and Stanat, 1989).

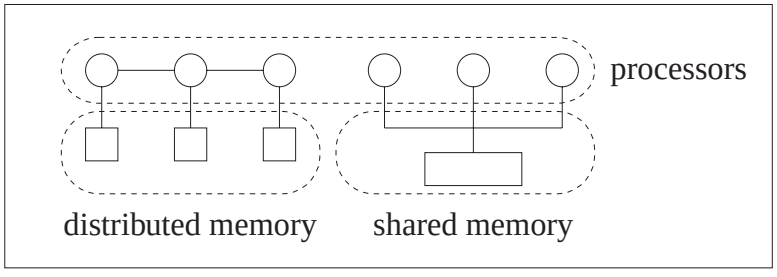


Figure 1-3: *Different types of architectures.*

Secondly - as we aim at developing a general purpose language - we decided to focus on Multiple-Instruction-Multiple-Data *(MIMD)* machines. These are also known as

asynchronous machines. They are more general than Single-Instruction-Multiple-Data *(SIMD)* machines (*synchronous* machines), which basically only support data-parallelism well.

The third restriction with respect to the architecture is that we will only consider parallel machines with *distributed memory*. These have the advantage that they are more scalable than *shared memory* machines, for which memory access constitutes a bottleneck. Nevertheless, we should note that shared memory architectures have the advantage of providing an easy programming paradigm, which is far less error-prone than the message passing model that is commonly used for dealing with distributed memory. Consequently, it is easier to realise a parallel implementation on an architecture with shared memory than on a distributed memory machine[3]. Our goal however is to eliminate the disadvantages of machines with distributed memory by providing an implementation of a functional language for these architectures, so that their greater scalability can be exploited in a suitable way, using the comfortable high-level programming model of functional languages.

1.4.4. Compilation versus interpretation

The quickest way to obtain an implementation is to construct an interpreter. It conveniently abstracts from the hardware details. This is its power and its weakness. The abstraction causes uncertainty about the behaviour of more sophisticated solutions based on code generation.

This is already problematic for sequential systems. Interpretation distorts the view on the real costs of computational steps. Not only because interpretation introduces overheads related to the interpretation itself, but also because interpreters have difficulties taking into account code optimisations for specific platforms. Consequently, it is difficult to derive the actual costs from interpretation.

For parallel systems, the problems caused by interpretation are worse. Interpreters tend to be one or two orders of a magnitude slower than running compiled code (One can see an example in chapter 4, table 4-5). Not only does this hamper the possibility of running complex (i.e. time-consuming) programs, but it creates a disparity in costs, as communication overheads are low compared to processing overheads. Typically, it is easier to obtain good speed-ups for interpreters running in parallel, than it is for compiled code. Thus, results from interpretation are easier to obtain, but also less revealing. Therefore we have chosen to construct a compiler.

---

3 Examples of shared memory implementations of functional languages are $< v,G >$ (Augustsson and Johnsson, 1989), AMPGR (George, 1989), GAML (Maranget, 1991), GRIP (Peyton Jones *et al*., 1987, 1989-b), Flagship (Watson *et al*., 1986, 1987, 1988) and HyperM (Barendregt *et al*., 1992). Examples of distributed memory implementations are PAM (Loogen *et al*., 1989), π-RED+ (Bülk *et al*., 1993) HDG (Kingdon *et al*. 1991) and SkelML (Bratvold, 1993).

1.5. The language that will be implemented: Concurrent Clean

The language we have chosen to realise all this, is Concurrent Clean. Detailed information on this functional programming language can be found in the book by Plasmeijer and van Eekelen (1993). In brief, it is a pure functional programming language that incorporates a number of features that are commonly found in other lazy, higher order functional languages. These include a Milner/Mycroft based polymorphic type system. However, as we will elucidate later, Concurrent Clean is not based on the *lambda calculus* nor on *term rewriting systems*, but on *graph rewriting*. This distinguishes it from other functional languages.

Originally, Concurrent Clean was designed as an experimental intermediate language. Its syntax was kept as simple as possible, allowing to concentrate on the essential implementation issues. As a result, it was possible to study a number of new concepts in a rather easy way. Examples of these are *term-graph rewriting* (Barendregt *et al*., 1987), *lazy graph copying* (van Eekelen *et al*., 1991), *uniqueness typing* (Barendsen and Smetsers, 1993, 1995-a, and 1995-b), and *strictness analysis* by means of abstract reduction (Nöcker, 1993-a).

This resulted in compiler technologies that allowed to generate efficient code for relatively small machines. The latter is rather exceptional, and as a consequence, people have started to use the intermediate language Concurrent Clean as a programming language, even though it was not intended as such. It became necessary to improve the syntax of Concurrent Clean, so that it conformed to a large extent to languages like Haskell and Miranda. The new Clean 1.0 system incorporates many high level features that were not present in previous versions. Examples are infix notation, overloading, type classes, list comprehensions, and support for arrays and records.

Unfortunately these features were not available at the time the research took place that is presented in this thesis. We were forced to write and test all programs in version 0.8 of the Clean system. However, to conform to the usual notations and to improve readability we have decided to use the syntax of version 1.0 for the program listings.

1.5.1. Graph rewriting

A Graph Rewriting System (GRS) represents a program as a set of graph rewrite rules. Starting with an initial graph, rewriting (reduction) takes place according to these rules. A part of a graph that matches the pattern of a rewrite rule is called a *redex*. Such a redex can be rewritten to *Root Normal Form* (RNF) by replacing it by an instance of the right-hand side of the corresponding rewrite rule. At any moment in time there will be multiple redexes, and therefore a reduction strategy is needed. Sequential strategies will at any point in time reduce a single redex, as opposed to parallel ones that will reduce multiple redexes at a time. The main difference with Term Rewriting Systems (TRS's) is that multiple occurrences of an identifier indicate *sharing* in a GRS, and not multiple copies of the expression itself.

In contrast to other functional languages, Concurrent Clean not only uses graph rewriting for its implementation, but also as its underlying computational model. In this way the difference between the implementation and the formalisation of the language is kept to a minimum. On the language level, we can see an example of this in the possibility

to indicate sharing of expressions explicitly. The importance of this becomes clear if we reconsider the necessity to reason about the operational behaviour of parallel programs in order to obtain efficient programs. The main drawback of using graph rewriting as a computational model, is that this field is still very young. Yet, the work of Barendsen and Smetsers (1992) has indicated that it is possible to reason in this formalism in a convenient way.

1.5.2. Uniqueness typing

As we already indicated earlier, ensuring referential transparency makes it rather hard to generate efficient code. In particular, functional languages cannot employ destructive updates on shared data structures: if one branch of computation would update a shared object in place, it would affect the outcome of the other branches that refer to the same object, and thus referential transparency would be lost. In contrast, a non-shared object may be updated in place, but unfortunately, a compiler cannot detect in general whether data structures are shared or not.

Concurrent Clean partially solves this problem by using its uniqueness type system (Barendsen and Smetsers, 1993, 1995-a, and 1995-b) to approximate actual sharing properties at compile time. This type system is able to derive - or enforce - that functions have exclusive (i.e. unique) access to certain arguments. If a function has such a unique argument it can safely overwrite the argument with the function result.

The uniqueness type system uses a unique type attribute to indicate that an object is not shared. Initially, each objects gets this attribute when it is created: only the creating function has a single reference to this object then. However, as soon as an object becomes shared - for instance by referring it twice in some expression - it looses its uniqueness type attribute forever. To avoid de-uniqueing too many objects, the type system takes into account the order of evaluation. For example, an object that is referred once in both the then-part and the else-part of a conditional expression will not loose its unique type attribute. At the same time, the formal arguments of a function may have a unique type attribute (this can be indicated by the programmer). If this is the case, the type system will either reject any application of such a function to a non-unique actual argument, or it will replace the function by an equivalent one that does not require uniqueness (a more detailed explanation of uniqueness typing can be found in chapter 6).

1.5.3. Annotations

Concurrent Clean enables the programmer to control parallelism explicitly by means of a few simple annotations:

- ***{I}***: This annotation starts up a new process on the current processor. It will reduce the annotated expression to *root normal form* (RNF). No communication takes place, instead, the new process shares the annotated expression with other processes. Processes on the same processor will run interleaved and scheduling is fair.
- ***{P}***: This annotation starts up a new process at some other processor (if there exists one) that will reduce the annotated expression to RNF.

- ***{P at** processor**}***: The same as ***{P}***, only now the expression 'processor' evaluates to a processor-id and the new process will be started at the given processor. Basically, all annotations are some form of the ***{P at** ...**}*** annotation.

Annotations cannot influence the outcome of a program, they merely change the order in which redexes are reduced, so one can deviate from the default *functional evaluation strategy*. In addition to the ones above, Concurrent Clean features a strictness annotation (an exclamation mark), which changes the evaluation order from lazy to eager. For clarity, annotations are distinguished by bold face throughout this thesis.

Clearly, the annotations listed above are very basic. This is their power and their weakness. On the one hand a suitable combination of these annotations will be able to describe virtually any parallel computation. So they are very general. On the other hand a single annotation does not have much expressive power. One may need many annotations in order to arrive at a useful parallel solution. In theory, the latter is not a very serious issue in a functional language. Using basic annotations one can construct skeletons, which are basically higher order functions that enable the easy use of certain parallel programming paradigms. However, such skeletons may not be generally applicable, nor very efficient, so they might not offer a very useful solution in practice.

Furthermore, this thesis will make clear that a serious problem lies in reasoning about the exact behaviour of these annotations. Even though the annotations themselves are very simple, it can be hard to apply them correctly. This has to do with the implicit communications in Clean. We will take a closer look at this below.

1.5.4. Lazy graph copying

Parallel implementations of functional languages can roughly be divided into two classes: those aimed at shared memory architectures, and those suited for architectures with distributed memory (see figure 1-3). The main difference between the two is that shared memory implementations will have multiple processors working on a globally accessible shared graph, whereas distributed implementations have to distribute the graph so that processors can reduce a private copy. To achieve this, a graph copying mechanism is needed. Important issues for shared memory implementations are the use of caches and locks. Conversely, implementations for distributed memory have to deal with the costs of graph transmissions. These areas of research are related in the sense that they focus on providing efficient access to graphs. Communication and caching are similar notions in this respect. On the other hand, transmission of graphs involves duplication, which sharply contrasts with the concept of sharing.

In principle, all communications in Clean take place implicitly, except for the ones implied by the annotations above. Whenever a function accesses an argument that is located at another processor, the corresponding graph will be transported automatically. So, communication is triggered by evaluation of remote arguments.

Copying a graph from one processor to the other can take various forms. The two extremes are formed by *eager copying* and *full lazy copying.* The former means that a graph is copied as a whole. This has the advantage that only a single message is needed to transmit a graph, which can reduce communication overheads. Full lazy copying on the other hand, will only transmit a node if it is needed at some other processor. Nodes are

copied one at a time on a transmit-by-need basis. Clearly, this introduces small messages and quite some protocol overheads, which can be costly if the whole graph turns out to be needed. On the other hand, if only few nodes are needed full lazy copying can be considerably more efficient than eager copying.

The main advantage of full lazy copying, is that it increases the possibilities for sharing of results as opposed to recomputing them. Full lazy copying retains references to the original nodes for as long as possible. On the one hand, this increases the chance of copying results, instead of work. And one the other hand, if a process needs a node that is being reduced by another process one can simply defer the copying process and resume it after the result has been computed. Programs that employ processes to consume the results of other processes rely on this way of copying graphs: instead of copying work back, a consumer will stop as soon as it hits a node on which a producer is still working.

Lazy graph copying combines aspects of both eager copying, and full lazy copying in order to avoid the copying of work (to some extent), while retaining the advantage of copying multiple nodes at a time. In principle, lazy graph copying copies nodes eagerly, unless it hits a *redex* that has a *defer attribute*. These nodes represent work that should not be copied. Copying stops at such a node and instead, a new reference to the deferred original is created (see figure 1-4). In this way, graphs can be copied partially. Once a former deferred redex has been evaluated it looses its defer attribute. Then it can be copied, but this will only take place on demand. Hence the name lazy graph copying.

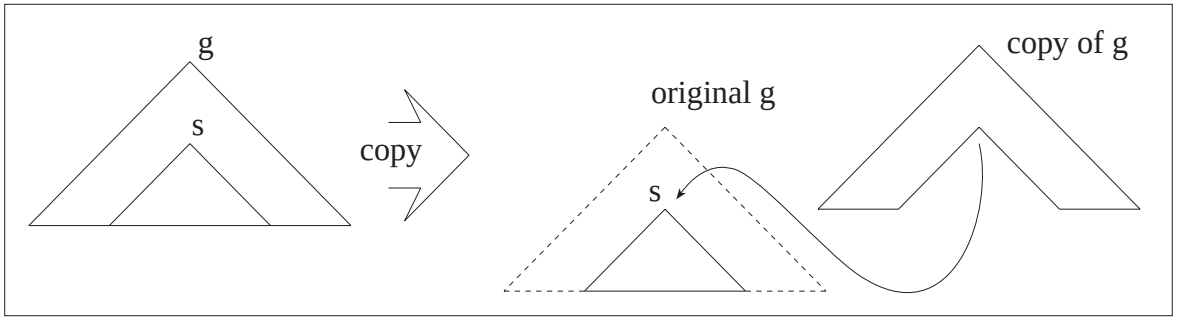


Figure 1-4: *The effect of lazy copying if a graph g is copied that contains a deferred sub-graph s (rooted by some redex). The sub-graph will not be copied. Instead, the partial copy of g will contain references to the original sub-graph. The deferred sub-graph may be copied later, after it has been reduced to Root Normal Form, but this will only take place if it is actually needed by some process that is working on the copy of g.*

To avoid deadlocks, one needs to ensure that any needed deferred redex will eventually be evaluated. Implicitly, Concurrent Clean only defers redexes that are being reduced, or are going to be reduced by a separate process (the latter constitutes the annotated redexes). It is not very useful to copy these redexes, as one would certainly copy work, and deferring them does not introduce problems, as they will be reduced. However, without further measures, one cannot not safely defer other redexes, because this could result in a deadlock. Therefore, the Clean implementation additionally guarantees that a new process will be started on any redex as soon as it turns out to be needed at another

processor. So, if a needed redex is deferred it will certainly be evaluated, and its result will be transmitted afterwards.

An important question in a lazy graph copying scheme, is which nodes should be deferred, in addition to the ones listed above. We will see that this greatly influences the way annotations are used, and that way we reason about parallel functional programs. In addition, at some level, the defer attribute is indispensable for certain structures that are part of the underlying operating system (these are commonly accessed by I/O functions). For these structures special rules apply, as they usually refer to physical objects such as disk drives, screens, and windows. For sequential programs it is sufficient to assign unique types to these objects if they should be updated in place (for example when writing to a disk). For parallel processing some extra rules are needed, because some structures are fixed to a particular location. Although sharing is possible, one cannot copy objects such as disk drives to another processor. At the lowest level these devices are immovable; they are essentially deferred. Using laziness, one can automatically create - and duplicate - references to such objects, but the objects themselves can be forced to stay at their physical location.

This indicates a direction that I/O systems may take in parallel implementations of Concurrent Clean. Operations on remote physical entities will not be able to access such objects directly, as they only have references to them. However, these references can be used to start up functions at the location of the remote objects. Chapter 8 presents some skeletons that may be used to accomplish this in an easy way.

1.5.5. The structure of the Concurrent Clean system

Compilation in Clean takes two phases. First, the functional Concurrent Clean program is translated to intermediate code for an abstract machine (the ABC machine, more about this later). This code is virtually the same for all architectures. From here, specialised code generators produce optimised code for different machines. This thesis will mainly be concerned with the last phase only - i.e. a code generator for a parallel machine -, because it is here that the specific problems with respect to a parallel implementation become apparent.

Code generation is only part of the story. A runtime system is needed as well. First of all, a runtime system allows easy reuse of code. And secondly, a runtime system offers an additional abstraction from the underlying architecture. This makes it possible to keep the code generator less hardware dependent. Consequently, the implementation can be kept relatively portable by constructing a sophisticated runtime system in a standard high level language (for technical reasons, this will commonly be an imperative one). Using this strategy, it turned out to be rather easy to construct a code generator for a parallel machine from a sequential implementation. The creation of an advanced runtime system on the other hand, was relatively complex. No such runtime system had been constructed before, so it had to be constructed from scratch, which turned out to be especially hard, due to lack of sufficiently powerful tools for the transputer hardware.

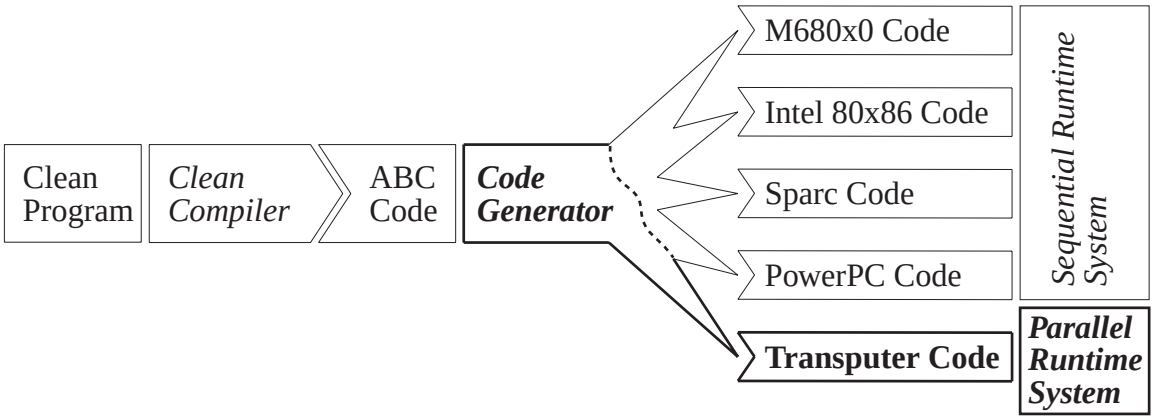


Figure 1-5: *The Concurrent Clean compilation traject. The italic type face indicates the components of the Concurrent Clean system. Plain text represents code. This thesis will merely address the implementation issues that are related to the bold part.*

1.5.6. The abstract ABC machine

Sequential implementations of Concurrent Clean (van Groningen, 1991; Nöcker, 1993-a and 1993-b; Plasmeijer and van Eekelen, 1993; Smetsers *et al*., 1991 and 1993) use an abstract machine that is very similar to the G-machine. This abstract machine - the ABC machine (Nöcker *et al*., 1991-b) -, is a stack based graph reduction architecture. It has three stacks: the Address stack, the Basic value stack, and the Control stack (hence the name). Furthermore, it contains a heap in which all graph reduction takes place. The use of these components will be explained below.

Graphs are basically collections of nodes that have some relation. In functional languages there is only one relation possible: a graph can be the argument of a node. Furthermore, there are only two types of nodes. They can either be redexes, or nodes in Root Normal Form. During graph reduction, redexes will be rewritten according to the reduction rules of the program, and eventually they will be overwritten by the resulting Root Normal Form. All this takes place in the heap.

The stacks are used to efficiently realise the proper reduction steps in the heap. The Control stack contains return addresses. It is used in a rather straightforward way to implement function calls. Both the A stack and the B stack are used for building expressions, and for passing arguments to functions or returning function results. The A stack refers to nodes in the heap, whereas the B stack contains values of a basic type, like integers and reals. The reason for having separate stacks for pointers and for basic values, is that this allows garbage collection without having to distinguish between both types of stack elements. Basic values can either be stored inside a node (and accessed via the A stack), or they can be placed directly on the B stack. Compared to the former, the latter allows more efficient manipulation of basic values, as nodes introduce more memory management overheads. Eager evaluation has the advantage that it often allows the B stack to be used for passing arguments and function results. However, this can only be allowed for strict arguments.

So far, we have only focused on sequential graph reduction. In contrast to these, parallel implementations of Concurrent Clean employ the Parallel ABC machine. This is basically an ordinary ABC machine with some extensions for parallelism. It is structured as a set of heaps that are interconnected by some communication mechanism. The graph is distributed over these heaps and at runtime an arbitrary number of processes (reducers) can be started to reduce some part of it. Each reducer proceeds as an ordinary ABC machine. It has an A, B, and C stack, and it is associated with one of the heaps. If it needs a part of the graph that is located at another heap, this part will be transported automatically (by means of lazy graph copying). Reducers that share the same heap run interleaved and they share the part of the graph that is stored locally.

A *locking* mechanism avoids that multiple processes reduce the same part of the graph. Each reducer first locks the redex it is going to reduce. As an effect, any reducer that subsequently accesses the locked node will suspend and put itself in the *waiting list* of the node. Later, when the locked node becomes updated with its result all reducers in the waiting list will be woken up.

1.6. The architecture on which we will implement Concurrent Clean: the transputer hardware

We have chosen to realise an implementation of Concurrent Clean on transputer hardware. This decision is based on two considerations. First of all, it was one of the few parallel architectures we had access to. And secondly, the transputer hardware matched the requirements we put forward earlier: it is a general purpose MIMD architecture with distributed memory.

Essentially, a transputer is just a processor (manufactured by INMOS). Several types of transputers exist. We have based our implementation on the T800 transputer. The main difference with ‘ordinary’ processors is that the transputer has hardware support for concurrent processing. It sustains processes at two priorities. *Low priority processes* are automatically *time-sliced* in a round-robin fashion by a hardware scheduler. Conversely, *high priority processes* are not time-sliced at all: they either run until completion, or until they have to wait for communication. Furthermore, each transputer has four hardware links. Instructions exist that enable a process to communicate with another one over such a link, or with another process on the same transputer. And finally, a transputer has a few kilobytes of on-chip memory (static RAM). The access time of the static RAM is 3 to 5 times smaller than the access time to ordinary (external) memory.

The transputer has six registers, but none of them is a general purpose register like the ones available in more traditional processors. The *workspace pointer* is a register that contains a pointer to the workspace of a process, which is just some part of memory. Addressing takes place relative to this pointer. The *instruction pointer* indicates the next instruction to be executed. The *operand register* is used to construct large operands. The remaining three registers (Areg, Breg, and Creg) constitute a tiny evaluation stack. They are the sources and destinations for most arithmetic and logical operations. Loading a value into this stack pushes B into C and A into B before loading the value into A. The contents

of C are lost. Conversely, storing a value from A into memory pops B into A and C into B. The value of C will be left undefined when this happens.

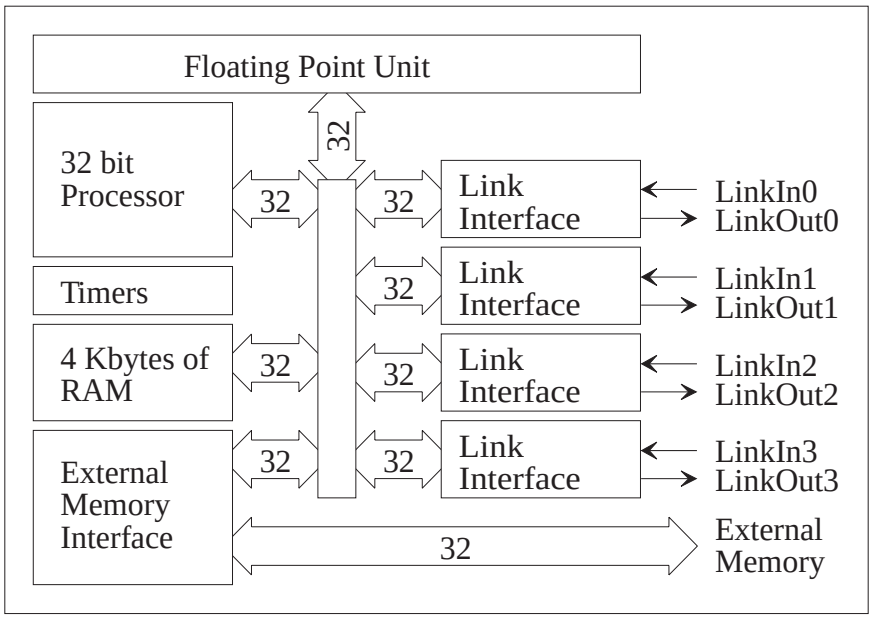


Figure 1-6: *The Transputer Processor*

One might expect that the transputer hardware would save the contents of this small register set during an automatic context switch. This however, is not the case. Only the instruction pointer and the workspace pointer are saved, and only a few instructions exist at which a context switch is possible. Consequently, context switches are uncommonly fast on transputer hardware (if we have a closer look, this is not entirely true. Having only a small evaluation stack one explicitly has to save the contents of registers periodically. In one way or another, the overheads of context switching remain. Even worse, on the transputer these overheads are also present during sequential computations. This is reflected in its relatively poor sequential performance).

An important advantage of the transputer hardware is its extendibility. One can create huge networks, simply by connecting the required number of processors. Communication bandwidth will not form a bottleneck compared to computational power, as both will increase equally fast. The only fundamental disadvantage is the notable delay that is possible during communications in a large network.

In practice, the transputer has a few additional drawbacks. Firstly, it uses a rather awkward model for sequential processing. In particular, it does not have any general purpose registers. This makes it relatively hard to use conventional code optimisations, which greatly depend on the existence of such registers. Secondly, the transputer does not have any protection hardware, nor debugging facilities. This makes program development a precarious task. Thirdly, the transputer primitives for parallelism are of a very low level. A lot of work is needed to get a sophisticated parallel system running. And finally, only one company manufactures transputer processors. It has not been able to keep up with the

`ldl offset`	*load (wsp + offset × wordlength) into Areg*
`stl offset`	*store Areg into (wsp + offset × wordlength)*
`ldnl offset`	*replace Areg by (Areg + offset × wordlength)*
`stnl offset`	*store Breg into (Areg + offset × wordlength)*
`ldnlp offset`	*add offset × wordlength to Areg*
`ldpi`	*add iptr to Areg*
`dup`	*duplicate top of evaluation stack*
`rev`	*swap Areg and Breg*
`mint`	*load $-2^{(wordlength\ -1)}$ into Areg*
`ldc constant`	*load constant into Areg*
`adc constant`	*add constant to Areg*
`add`	*add Areg and Breg*
`sub`	*subtract Areg from Breg*
`mul`	*multiply Areg with Breg*
`and`	*logical and of Areg and Breg*
`gt`	*Test if Breg > Areg and place result in Areg*
`cj label`	*jump to label if Areg equals zero*
`j label`	*jump to label, context switch possible*
`gcall`	*jump to Areg, and store return address in Areg*
`runp`	*start a process, of which Areg contains the wsp*
`stopp`	*stop the current process*

Figure 1-7: *A list of some important transputer instructions and their meaning. In this list '(address)' stands for the contents of 'address'. Only the unconditional jump instruction may cause a context switch. For all instructions holds that loading a value into Areg pushes the original value of Areg into Breg, and the value of Breg into Creg. The opposite happens when a value is popped from Areg and stored into memory: the contents of Breg moves to Areg, the contents of Creg moves to Breg, and the contents of Creg becomes undefined. Dyadic operations first pop their operands (Areg and Breg) and push the result in Areg.*

leading industries. As a result, the transputer architecture has become rather slow compared to modern processors.

1.7. An overview of this thesis

In essence, this thesis explains major design decisions and it reports the behaviour of an implementation of Concurrent Clean that we have realised on transputer hardware. It does not show the efforts that were involved in developing this implementation. Much hard labour is not in this thesis, but in source code. This section will give a short overview of the work we *will* reveal in the remaining chapters.

This thesis can roughly been divided into two parts. The first part (chapters 2 through 5) mainly focuses on implementation issues in a bottom-up manner, starting with low level topics, and ending with high level ones. It shows what techniques are needed to support the

annotations for parallelism as presented in section 1.5.3. In contrast, the remaining chapters (chapter 6 and higher) will rather take a look at using these annotations for creating efficient parallel programs and they will evaluate the techniques that have been presented earlier.

However, we should keep in mind that such a division is not a very strict one. The development of efficient implementation techniques requires a constant evaluation of these techniques, while each evaluation commonly leads to new implementation methods. During the development of implementation techniques, these become ever more sophisticated, and the point of interest gradually shifts from machine level issues to language level issues. Problems shift from technical to fundamental.

Most information in this thesis has been introduced earlier in a number of articles. These have been presented at international workshops and conferences. Some chapters (2, 6, 7, and 8) are adapted forms of these papers. The others have been written from scratch. This was necessary as the information in the papers themselves was largely outdated. We will have a closer look at all chapters below.

1.7.1. Chapter 2: packet routing

Chapter two shows a technique for realising efficient communications between any two processors in a transputer network. Important aspects are the avoidance of deadlocks and starvation. This routing mechanism forms the basis of the transputer runtime system we implemented to support Concurrent Clean. This chapter has been based on two papers: *the Class Transputer Router*, presented at the PaCT conference in Obninsk, Russia (Kesseler, 1993-a), and *Efficient Routing Using Class Climbing*, presented at the World Transputer Congress in Aachen, Germany (Kesseler, 1993-b).

1.7.2. Chapter 3, 4, and 5: the implementation chapters

Chapter three will explain the basic structures of the implementation. It will show the realisation of the stacks, the heap, the processes and the nodes. Chapter four is about code generation. It will make clear that ordinary sequential code generation for register-based architectures can be extended to parallel code generation for the transputer. Chapter five focuses on management of graphs. This comprises the implementation of garbage collection and lazy graph copying. It will become clear that these are rather complex issues.

Part of the information in chapter 3 to 5 can be found in *Implementing the ABC machine on Transputers*, which has been presented at the PCA workshop in ISPRA, Italy (Kesseler, 1990), and in the articles that have been presented at the implementation workshops of 1991 and 1992 (Kesseler, 1991 and 1992). However, this information is largely outdated, so we have chosen to write these chapters completely from scratch.

1.7.3. Chapter 6: The influence of Graph Copying on Runtime Semantics and on Uniqueness Typing

Graph copying is an extension of standard graph rewriting semantics, while uniqueness typing has been defined for the standard semantics only. It turns out that a conflict exists between certain forms of graph copying and uniqueness typing. This chapter will identify the problems and propose a solution that allows the combination of graph copying and

uniqueness typing. This solution not only allows the use of uniqueness in parallel programs, but it also provides very clear runtime semantics that make reasoning about parallel programs much easier. The source of this chapter is formed by *Lazy Copying and Uniqueness - Copyright for the Unique*, which has been presented at the implementation workshop in Norwich (Kesseler 1994-a).

1.7.4. Chapter 7: The Costs of Graph Copying

This chapter shows the importance of uniqueness typing for parallel implementations. Without it, arrays cannot be used, and these turn out to be very important to reduce graph copying costs. We will see that the same algorithm can have very different behaviour, depending on the data structures that it uses. This shows the importance of clear runtime semantics. Chapter seven has been based on *Reducing Graph Copying Costs - Time to Wrap it up*, presented at the PASCO conference (Kesseler, 1994-b) in Linz, Austria.

1.7.5. Chapter 8: Constructing Skeletons

The last chapter shows that efficient skeletons can be constructed from the primitive annotations that Concurrent Clean provides. Considerable speed-ups have been obtained this way, while the programs themselves remained simple. However to accomplish this, we needed thorough knowledge of the runtime semantics of Clean. It also turned out to be necessary to introduce a new function to allow the precise placement of functional processes. In this chapter we make elaborate use of the improved runtime semantics that have been introduced in chapter 6. This chapter has been derived from *Constructing Skeletons in Clean - The Bare Bones*. This paper has been presented at the High Performance Functional Computing Conference in Denver, Colorado (Kesseler, 1995).

2. Packet Routing

Efficient - and reliable - communication forms the basis of a parallel system with distributed memory. To obtain an implementation of Concurrent Clean one needs to ensure that arbitrary processes can transmit messages to each other. Unfortunately, the transputer architecture does not provide this in hardware. A transputer processor can only communicate directly with its four neighbouring processors. Thus, we need some message passing software.

This chapter concentrates on a general store-and-forward routing algorithm that is adaptive and avoids deadlocks and starvation. The algorithm is based on the one used by the communication processor of the Decentralised Object Oriented Machine (DOOM) architecture, which uses class climbing to avoid deadlocks. It has been altered in such a way that it can be used for the transputer hardware. The changes made to the original algorithm will be presented and their correctness with respect to avoidance of deadlock and starvation will be clarified. Performance figures clearly indicate that the resulting class climbing algorithm can compete with - and often outruns - other solutions.

This chapter has the following structure. Section 2.1 will explain the necessity for developing a software routing mechanism on the transputer architecture. Section 2.2 will give a description of the original DOOM algorithm, on which we will base our solution. In section 2.3 will show that this algorithm cannot be used for a network of T800 transputers unmodified and it will proceed with the presentation of a modified algorithm, called the Class Transputer Router (or CTR). Section 2.4 will introduce some optimisations that we employed to improve performance. In section 2.5 we will present performance measurements for the CTR, and we will relate these to performance figures of other routing mechanisms for the transputer architecture. And finally, we will present our conclusions in section 2.6.

2.1. The need for a routing mechanism on transputer hardware

Clean allows processes to be started at arbitrary processors. In addition, - as we will see in chapter 5, references to remote graphs may travel through the network. As a result, a reference can point to a graph on *any* processor. Furthermore, the size of graphs is not known in general. In theory, it may be anything between a few bytes and the entire memory space of a processor. In practice their size commonly varies from some bytes to several kilobytes. Consequently, there is no general pattern for the routes and distances a message may travel, nor of the size it may have. This implies that we need the possibility to send

messages of any size between arbitrary processors. In addition we need reliable communications to make debugging possible, and to enable the incorporation of I/O facilities.

Unfortunately, the transputer does not provide such a communication mechanism in hardware. The T800 can be connected directly to no more than four others. Thus, networks of more than five transputers are not fully connected. At the same time it only supports instructions to communicate with neighbours. Messages between two transputers that are not directly wired need to be routed via some intermediate transputers by means of some software solution.

Some routers have been developed for the transputer, but many of them do not consider problems of deadlock and starvation thoroughly. Those that do sometimes use a lot of memory, limit network topology - for instance by disallowing cycles in routing paths -, or they are inefficient at higher network loads. Apart from these drawbacks not all routers (in particular their sources) are easily obtainable. So, when we started implementing Clean on the transputer we faced the situation that no routing software existed that fulfilled our needs. A lot of research has been devoted on a theoretical level to avoidance of deadlock and starvation in packet switched networks, without designing and testing concrete algorithms on concrete machines. Consequently, we had to implement a routing mechanism ourselves. It had to meet the following requirements.

- asynchronous point-to-point communications of messages of any size between arbitrary nodes (similar to a global block move).
- absence of deadlock
- absence of starvation
- independence of network topology and size
- efficient usage of buffer space
- high data throughput

In a massively parallel system with distributed memory delays will be relatively high. This cannot be avoided. It is a natural characteristic of a large system, in contrast to good data throughput, which can be sustained rather well. If we consider a store-and-forward system like a transputer network, keeping delays constant would require a fully connected network, which is $O(n!)$ for a network of size n. Conversely, maintaining the data throughput for a connection between two nodes means we have to reserve (the capacity of) a link for every hop a message takes. This is related to the diameter of the network, which can be as low as $O(\log(n))$. On average, for a busy network we need $O(n \cdot \log(n))$ links. A hypercube satisfies this requirement. In practice, a lower connectivity suffices, as networks are seldom used continuously at their maximum capacity.

The delay of packets has to be kept as small as possible, but - assuming that delays are unavoidable - Concurrent Clean has been designed to deal with delays at a higher level, by means of annotations (see also section 3.1). Delays are part of the costs of delegating work. The negative effects (i.e. processes waiting for some result) can be dampened by running multiple processes on the same processor, so that useful work can be done while some processes are waiting. In addition, strictness and speculative evaluation allow communications to take place before they are actually needed (in contrast to pure lazy

evaluation). We must note however, that it still remains to be seen whether such techniques are sufficient to avoid the effects of delays in serious programs.

The requirement to send messages of arbitrary size is a little too strong. A well-known way to deal with large messages, is to split them up in smaller parts that are transmitted separately and reassembled at the destination. These parts - i.e. packets - will not exceed a certain maximum size, which makes low level routing less complex. In addition, it decreases delays as it does not require the whole message to arrive before it can be passed on. The pieces of a single message are pipelined. Apart from this message routing and packet routing are essentially the same. We will focus on packet routing, because of its simplicity.

Various ways exist to implement such a router. We present one using buffer classes. It is based on the adaptive routing algorithm that was used for the special purpose router processor of the Decentralised Object Oriented Machine (DOOM) architecture (Annot and van Twist, 1987). The algorithm had to be altered in order to be usable for a network of transputers. As it uses a class climbing algorithm to avoid deadlocks we have called our router the Class Transputer Router (CTR for short). In this thesis we will refer to it as the Clean Router as well, to avoid confusion between the original algorithm and the new one.

We will see that adaptive routing (also called dynamic or datagram routing) is better than routing over fixed paths (or virtual circuit routing) when the network load is high. It not only avoids hot spots, but it also increases the throughput for broken-up messages by using multiple paths for a single message. Unlike fixed routing, it tends to scatter the pieces of a message over the network.

2.2. The original DOOM routing algorithm

The algorithm that has been used to implement the Clean router is based on the one presented in (Annot and van Twist, 1987). It has been developed as part of the DOOM project and we will refer to it as the DOOM router. It has been proved to prevent deadlocks, provided that each packet that reaches its destination is consumed within a finite amount of time. To accomplish this, a fixed number of buffers and a class-climbing algorithm are used. Starvation is avoided by a mechanism that passes around privileges for buffers. It is adaptive, i.e. it does not use a fixed path to route packets.

2.2.1. The components of the DOOM router

In the original algorithm each communication processor has a number of connections to other communication processors. Each connection consists of two physical links (wires) and each processor has two communication processes per connection: an input process and an output process. Each output process is connected to an input process on another processor via a dedicated physical link (see figure 2-1).

Every physical link is able to carry messages in either direction but not in both directions at the same time. Part of these messages contain control information to regulate communication; The rest consists of data messages that contain the actual information that has to be transported. Data messages are not allowed to exceed a certain maximum size and

will be referred to as data packets, or packets in short. Packets always travel from an output process to an input process.

In addition to normal data, each packet contains its class and its destination. These are the only two values within a packet that are used by the router. The class is used to avoid deadlocks; packets with higher class have more privileges. This will be explained later. The destination is used to determine over which links a packet should be transmitted. The router uses a routing table for this: it indicates for each destination which links can be used for transmission.

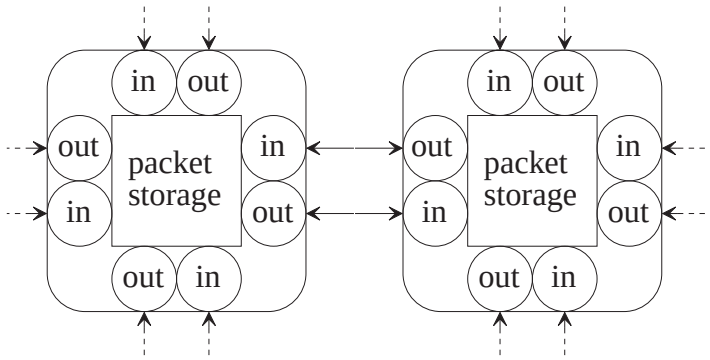


Figure 2-1: *DOOM communication processor overview.*

2.2.2. Process interaction

The communication processes repeat a small number of actions infinitely, as depicted in figure 2-2. They operate quite independently. On a single processor they interact by means of a shared packet store to which they all have mutually exclusive access (figure 2-1).

The packet store consists of a fixed number of fixed size buffers. The buffers are structured as a number of output queues. For every output process such a queue exists. Each is allowed to pick packets out of its own queue to transmit over its link. However, buffers are not dedicated to a certain queue. The same packet may in fact reside in distinct queues simultaneously, so that it is shared among several queues; it is not copied physically. This indicates that the packet can go in different directions to reach its destination. Whether it will actually be transmitted over several links depends on the kind of communication. For point-to-point communications it will be sent over just one link, namely the one that is able to send the packet first. Using broadcast routing it will be transmitted over several links without replicating it in the packet store.

The communication processes continuously perform the following actions. Whenever an input process is able to reserve a buffer of a certain class c, it sends a request over its link. The corresponding output process then starts searching in its own output queue for all packets that match this request, which are all packets of class $\geq c$. From these packets it takes the one that has arrived first. In other words: the criteria to arbitrate between packets are time of arrival and class, where the class of the chosen packet is not necessarily the highest one. After a packet has been chosen the output process removes it from all output queues it was placed in and sends it over the link to the requesting input process. If no packet could be found it sends a cancel message. On reception of a cancel message the

input process frees the buffer. Conversely, when a packet arrives it is stored in the reserved buffer and put in all output queues that are indicated by the routing table.

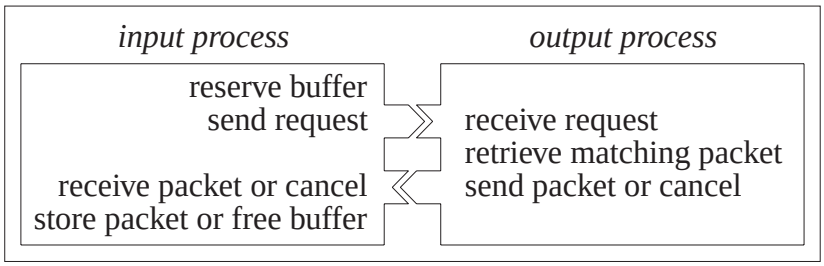


Figure 2-2: *Actions of communication processes.*

The initiative for the transport of a packet is taken by the receiving process, not by the sending one. As a consequence, one can postpone the decision to which neighbour each packet should be forwarded until it is certain that at least one neighbour is willing to accept the packet.

2.2.3. Class climbing

Class climbing is a well-known method to avoid deadlocks (Bertsekas and Gallager, 1987, Stallings 1988). In essence, it uses priorities on packets to avoid deadlocks (packets are put in a certain class). It can be used for routing algorithms that are able to force a packet to travel over a path without any cycles, i.e. a packet may travel in cycles temporarily, but in critical situations it must be possible to reach the destination via a non-cyclic path. Like some other deadlock-avoiding algorithms class climbing is based on the consideration that deadlocks are the result of cyclic (buffering) dependencies between different messages. Unlike others that limit the number of possible physical paths a message may travel it avoids these dependencies by means of sophisticated buffer management. The DOOM algorithm uses this method and its correctness has been proved (Annot and van Twist, 1987).

As the name suggests a class climbing algorithm works with classes: at any moment in time each packet has a certain class. During transportation this class may be increased, but never decreased. Buffers have a class as well and for every possible class at least one buffer exists. Packets of class n can pass through buffers of class n or lower only. This means that packets of a higher class have more buffers available to travel through than packets of lower class.

In some class climbing algorithms a packet gets a higher class each time it hops over a link. This implies that the minimum number of buffers that must be allocated is equal to the distance a packet can travel, which is related to the diameter of the network. The DOOM router uses a different approach. It superimposes an acyclic graph on the physical communication network. This means each connection gets a direction. As long as a packet travels along - respectively against - this direction its class remains the same, but when it changes direction its class is increased. This means that the number of buffers can be limited considerably for networks that have few cycles but a relatively large diameter (e.g. an array, a tree, or a ring).

The number of buffers needed is linear with the network diameter at worst. This may seem substantial (Son and Paker 1991), but one should keep in mind that large network can be constructed with a diameter of *O(log n).* This increase in buffer space does not constitute any problem in practice. It is rather the size of the routing tables that is more worrying, as this grows linearly with increasing network size, unless special measures are taken, such as grouping routing information for certain areas in the topology.

2.2.4. Buffer management

In order for the algorithm to work efficiently the request messages, which have been mentioned above, should always request packets for as low a class as possible. A rather intricate administration is used to accomplish this. One of the things that is carefully avoided is copying the contents of buffers to others, by connecting buffers to a certain class only logically, never physically.

An example will clarify the disadvantages of buffers that are physically connected to a certain class. Suppose a processor has two buffers, one of class zero and one of class one. Both are filled with a class one packet. When the class one buffer is freed the router should be able to send a request for a class zero packet, but there is no buffer available to store it in. The router does not solve this by moving the class one packet from the class zero buffer to the class one buffer, but by adjusting its administration to change the logical association of the buffers with a certain class. In the rest of this paper we will talk about a buffer of class *x*, instead of a buffer that is logically associated with class *x*.

It is important that buffers are dedicated to certain classes or links as little as possible. This allows for the most efficient utilisation of resources under all circumstances. Class zero buffers are most general as they can be used for packets of any class. For this reason every class other than zero has only a single buffer associated with it, and the remaining buffers are associated with class zero. This is reflected in the buffer administration, which is split in two parts: a zero class part and a non-zero one, enabling the use of a more efficient administration for zero class buffers. Next, we will see the same holds for the fairness administration that prevents starvation.

2.2.5. Starvation

In addition to deadlock avoidance the DOOM algorithm provides a method to prevent starvation. This is done in a reasonably straightforward way. As noted above the buffer administration is split in two parts: for zero class buffers and for non-zero ones. For both kinds a different method of fairness administration has been devised.

Zero class buffers can be allocated by any input process as long as there are more zero class buffers available than the number of input processes. When fewer buffers are available they are distributed according to an array of Booleans. When the n^{th} Boolean is true input process *n* may allocate a zero class buffer. If it does so, it should clear the corresponding Boolean, thereby throwing away its privilege. On freeing a zero class buffer, a round-robin method is used to reset one of the cleared Booleans.

Each non-zero class buffer is dedicated to exactly one input process at any moment in time, which means that only this input process may allocate a buffer of that particular class. After it has used its privilege the input process passes it on to the next input process. It

should be noted that this assumes that each input process never suspends itself, so that privileges are passed around quickly and never get 'stuck' at a sleeping process.

To be clear, these methods to avoid starvation are not needed to ensure absence of deadlock. The class climbing algorithm alone suffices. This is important, as we will see later we had to adopt a different fairness administration for the CTR.

2.3. The modified routing algorithm: the CTR

The DOOM algorithm has been designed for a special router processor. It is not possible to implement it directly on transputer hardware (T800), but we have conformed the CTR to the original algorithm as much as possible. This section will focus on the problems that are related to a transputer version of the algorithm and the solutions we have adopted.

2.3.1. The incompatibilities between the original algorithm and the transputer hardware

Two aspects of the DOOM algorithm need careful consideration. First of all it relies on inter-processor connections that consist of two independent links each. Secondly the input process is not blocked ever: when it receives a cancel message it goes on requesting and when it cannot reserve a buffer it starts polling for one. The output process is blocked only at moments in which it waits for a request to arrive. In practice this means it waits only during the time the corresponding input process is polling for buffers. This polling is rather exceptional: normally both processes are continuously running. These aspects introduce two important problems when implementing this algorithm for a network of transputers:

- Although each T800 transputer connection consists of two separate links (wires), these links cannot operate independently and a single link cannot send messages in both directions: each transputer connection consists of two unidirectional links. When data is sent over a link, the opposite link is used for acknowledgements. This imposes a problem as a single link cannot be used by multiple processes simultaneously. If an input process sends a request and an output process (at the same transputer) sends a packet over the same link at the same time the transputer link hardware will not work correctly.
- On the transputer the router processes should run at high priority. Not only does this facilitate the use of shared data structures - high priority processes are not time-sliced - but much more important: the use of high priority processes is vital when good response times have to be achieved, as noted in the ZAPP experiments (Goldsmith, McBurney and Sleep, 1993). In the original DOOM algorithm the router processes run without any interruption. Running these processes at high priority on the transputer would not allow any low priority process to run.

To overcome the problems mentioned above the algorithm has been changed in two ways. In the first place the messages from the input processes (requests) and the messages from the output processes (packets and cancel messages) that have to be sent over the same transputer link are multiplexed over that link. Secondly the router processes will suspend themselves in certain situations. The latter solution introduces an additional problem that is

related to fairness administration: processes might fall asleep while holding some privilege and this would prevent others to access certain buffers. We will take a closer look at these problems below.

2.3.2. Multiplexing of messages

The multiplexing of messages is done as follows. To avoid inappropriate use of transputer links output processes are not allowed to read from a link and input processes are not allowed to write to a link. Instead each input process delegates write actions to the corresponding output process and when it receives a message for an output process it passes it on (see picture below).

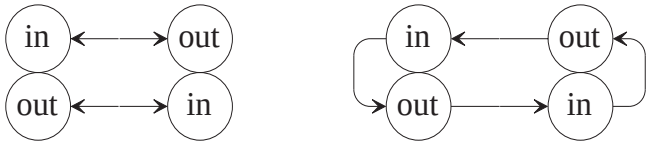


Figure 2-3: *The DOOM links (left) versus the CTR links (right).*

As can be seen in the picture above, request messages now have to travel through two intermediate communication processes to reach their destination. This does not introduce a performance penalty as messages from an input process to a local output process are not copied in reality. They are globally known to both processes.

It will be clear that this change of inter-process connectivity may not influence the original behaviour of the algorithm. Two issues need to be considered. First of all, the ring of unidirectional links in the picture above should support two entirely separate data streams. Processes must accept any mixture of messages from the two streams: no communication process may at any time commit itself to accept only one kind of message. Secondly, the use of a ring introduces the possibility of a classical store-and-forward deadlock. Such a deadlock arises when all processes in the ring start to send a message while none of these processes is able to accept a message. If we examine the original DOOM algorithm however, we can see that there are never more than two messages travelling simultaneously through the ring, so a store-and-forward deadlock cannot occur.

2.3.3. Sleeping processes

In DOOM, the router processes are running even when there are no packets to be transported: the input processes are continuously issuing requests and the output processes cancel these. As explained above this is a serious problem for a transputer implementation. One would like to suspend the routing processes when there is no message to be sent. However, suspended processes do not pass on privileges, so we will abandon the original fairness administration and introduce a different one later. Now we can suspend processes at some moments. We have implemented the following solution.

When an input process receives a cancel message for a certain class this means that the other side of the connection has no packet available that matches the previously issued request. Because of this it is no use to go on requesting packets of this class or higher. If no request of lower class can be sent the input process falls asleep instead of issuing useless

requests. It will be woken up in two cases: (a) when the input process is able to send a request of lower class than the one cancelled and (b) when the other side tries to send a packet that matches the previously cancelled request.

The Clean router realises this as follows. When a cancel message is returned the input process 'falls asleep'. It does not really stop itself though, as it should continue reading messages from the link. In this case 'falling asleep' means that it frees the buffer it has reserved and withholds a new request message, preventing a subsequent cascade of cancels and requests. There are two ways to reintroduce a request in the network. First of all, on the processor that contains the 'sleeping' input process, each time a buffer becomes available of lower class than the one that has been cancelled, the output process is notified to issue a new request on behalf of the input process (as the input process itself might be waiting for input). Conversely, when the output process on the 'other side' gets a packet that matches the class of the previously cancelled request, it sends a special 'wake-up' message (a signal message) to the 'sleeping' input process to make it send a new request. Each signal contains the highest class of all messages that match the previous cancel. This is used to avoid sending superfluous requests: when a signal arrives, while no suitable buffer is available for this higher class either, the input process does not wake up. In effect each class n signal simulates a request of class *n+1* followed by a cancel. It just increases the class that has been cancelled and only if this means a 'matching' request can be sent, the input process is woken up.

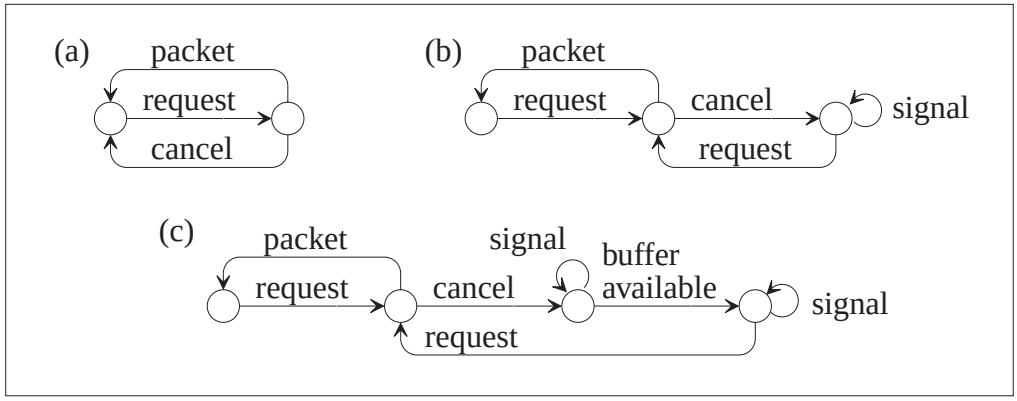


Figure 2-4: *State transitions of the router processes. 'Buffer available' means a buffer can be allocated of lower class than the one cancelled. (a) represents both the input process and the output process in the DOOM algorithm. (b) and (c) depict processes of the CTR. (b) stands for the output process and (c) models the input process. (b) is an abstraction of (c): signal* · buffer available · signal* is represented by signal* as freeing a suitable buffer is invisible for the output process. (a) is an abstraction of (b) - and therefore of (c): signals are non-existent in the DOOM algorithm. Note that a signal may be sent to an input process that has already been woken up: at some point a signal is possible while the input process is trying to send a request.*

If we abandon the original method to avoid starvation it is easy to show that suspending the input process does not violate the mechanism of the original algorithm to

avoid deadlock. Two scenarios are conceivable after an input process has fallen asleep: (1) it will be woken up in the future and (2) it will not. We will take a look at both below.

It is trivial to show that the first scenario is safe: when the input process wakes up it proceeds at the exact point it had fallen asleep. This can simply be seen as a case where the input process runs extremely slow for a while. As process speed plays no role in the proof of the original algorithm, the first scenario is perfectly safe.

Suppose the second scenario occurs, i.e. an input process falls asleep and is never woken up again. Consider that a suspended process does not hold any shared resources: it has previously freed the buffer it had reserved, and it does not hold any privileges as no fairness administration is used. Furthermore, it continues passing messages from other processes. This means that it does not influence any other process except the output process on the neighbouring processor, which will not get new requests.

Let us assume that at some point in time the input process stopped due to cancellation of a request of class n and no signal of a class $\geq n$ will be received in the future (note such a class exists). This implies that (a) the processor that contains the input process will never free a buffer of class $< n$ and (b) the output process at the other side never has a packet available of class $\geq n$. In the original algorithm (a) implies that all future requests will be of class $\geq n$, whereas (b) means these can only be answered by a cancel as there will never be matching packets at the other side. In other words, the DOOM algorithm would request packets for an infinitely long time. This is equivalent to never waking up the input process.

2.3.4. Starvation

Although it is possible to build a deadlock free router with the techniques presented above, they invalidate the DOOM method to avoid starvation. To be more specific: if a privilege is granted to a sleeping process it might not get passed on to others quickly. Even worse: it may not get passed on at all. This has led to a different fairness administration for the CTR that does not allow sleeping processes to have any privileges.

To avoid starvation it suffices to make sure that each process eventually gets buffers of arbitrary low class within certain needs: sleeping processes need a buffer of lower class than the one that has been cancelled, whereas processes that are awake are in need of a buffer of any class. In contrast to the original algorithm. the CTR passes around privileges for all buffers according to the needs of processes. When a buffer of a certain class is freed the privilege for this class is passed to the next process that needs a buffer of this class. If a process is found that already has a privilege, the process keeps the privilege for the lowest class and passes on the privilege for the other one. If no process needs the buffer it becomes free to be allocated by any process. If an input process receives a cancel for class n while it has a privilege for a buffer of a class $\geq n$, it passes on this privilege.

This means that a process never has more than one privilege: it keeps the best privilege it gets. This is sufficient to avoid starvation and it allows to efficiently pass on the privilege when a process falls asleep (one does not need to pass on many privileges). Note that a sleeping process cannot get any privileges without being woken up as well, which avoids the problem of the original fairness administration.

Using induction over the classes it is easy to show the original algorithm avoids starvation. To prove the new algorithm, one needs to add that while an output process tries

to forward a packet of class *n*, the input process on other side will never get a cancel of a class ≤ *n* (respectively *n+1* if the class should be increased), and thus will stay eligible for a privilege of a class n buffer (respectively a class *n+1* buffer).

2.4. Optimising the CTR

The routing algorithm that has been presented so far prevents deadlocks and starvation, but it contains some inefficiencies. In this section a few optimisations will be presented that improve the execution speed of the router. In general these optimisations work best at higher loads of the network.

2.4.1. Variable packet size

To avoid having to send a lot of bytes for small packets, packets have variable size up to some maximum. It is not entirely trivial to implement this on a transputer as the communication hardware requires both sender and receiver to agree on the number of bytes that have to be transferred. For this reason the router splits packets in two parts. The first part has a fixed length and contains four fields: the kind of the message (to distinguish packets from requests, cancels and signals), the class of the packet, the length of the packet, and finally a request field, which will be explained below. This header layout is not only used for packet headers, but also for signals, requests and cancels. In the following we will use the word ‘header’ to indicate both packet headers and control messages. In contrast to control messages, packet headers are followed by a body of the exact length that is mentioned in the header.

2.4.2. Overlapping data and control messages

Having received a packet header, one does not have to wait for the packet body to return a new request or to issue a signal to another processor. The CTR overlaps the sending of a new request with the transmission of a packet body. In this way the next request will have arrived by the time the current packet has been transmitted entirely (if it is not too small). This minimises the delay between subsequent packets. Signal messages are not (yet) overlapped with incoming packet bodies. It would impose some problems for large packets. It will trigger an early request for the packet and this may arrive when the packet body has not yet been entirely received. This would result in additional cancels and signals unless the router becomes more complex.

2.4.3. Combination of messages

On the transputer it is more efficient to send a small number of large messages instead of a large number of small ones. On each connection two opposite data streams are implemented by the router. Packets, cancels and signals are transported along the direction of data and request messages go against it. The router uses this to piggy back requests of one data stream on the non-request messages of the other stream. For this the request field of each header is used: it contains the class that is requested, in contrast to some special value if no request is piggy-backed. Due to this optimisation no control messages are needed when a connection is used continuously in a bi-directional way: only packets

containing piggy backed requests are needed (in contrast to the worst case overhead when traffic is low: the sequence signal - request - packet - request - cancel may be needed, which contains four additional - but small - messages)

2.4.4. Traffic control

When a lot of buffers are in use the router is slowed down due to a decrease in the number of links that can be used simultaneously (and bidirectionally), resulting in sub-optimal throughput. One can picture such a situation as each processor having only one hole to receive a packet in (instead of at least one per link). At any moment in time only one neighbour can fill such a hole, thereby transferring the hole to itself. If no measures are taken, this situation might persist for a very long time. A slowdown of a factor of 2 has actually been measured during tests of an early version on a ring where this situation was created artificially. Due to fairness administration in combination with a heavy load it did not resolve at all. A heuristic has been used to suppress this problem: fairness measurements are related to the number of free buffers. When a lot of buffers are in use at some processor, new packets are accepted less frequently at this processor, favouring through-routing and delivery of packets. This increases the overall number of routing-holes in the network.

2.4.5. Potential improvements

Some additional changes to the algorithm may improve performance a little more. These have not yet been implemented, nor tested, mainly because it is not clear whether these are worth the implementation effort.

First of all, additional tests for waking up processes might increase performance. Currently, processes are woken up eagerly, i.e. whenever there is a possibility they can do something useful. This may not be the case. Sometimes processes are woken up so early that they fall asleep immediately thereafter. It is not clear whether the costs of starting and stopping processes outweigh the costs of performing these extra tests.

Furthermore, some techniques may decrease control overheads in case the load is not extremely high. First of all it may be better to wait a little before returning a cancel message. A suitable packet might arrive shortly. This avoids an additional cancel - signal - request sequence. One cannot not wait too long, though. Not only should the requesting process free its buffer within a reasonable amount of time, but meanwhile, it may have some lower class buffer available as well. Secondly, when a packet arrives that can be sent over multiple links, all these links are currently woken up if they are asleep. This means several signal messages are generated, which is rather superfluous when traffic is low. A single one is probably sufficient. If this does not result in a suitable request within some time other links could be signalled after all. For high loads this is somewhat inappropriate as one would like to activate as many links as possible in this case.

2.5. Performance measurements

In this section we will present the results of some tests that were carried out with the CTR on a network of transputers. These will be compared with tests of the Helios

communication mechanism and the Parix communication mechanism. Helios and Parix are both commercial unix-like distributed operating systems for the transputer. It will be shown that Helios not only is less reliable - it is not deadlock free -, but that it is slower as well, in particular for larger networks and higher traffic loads. The tests of the Parix communication primitives had to be carried out on a different transputer architecture - i.e. with different clock speed and interconnection hardware - and must be treated with some care. Parix is very fast for small messages, but relatively slow for large messages, especially for larger networks and more complicated communication patterns. Moreover, we did not succeed in running all tests with Parix. At the moment of testing, we did not have enough information about the performance of other routers under the same circumstances to relate these to the CTR properly.

For the Virtual Channel Router (VCR) some figures about message latency over 1, 2 and 3 hops in a quiet transputer network have been published (Debbage *et al.*, 1991). For small messages in a quiet network the VCR outperforms the CTR. This is partly caused by differences in raw link speed. Looking at the latency for messages of different size one can deduct the raw link speed is approximately 1500 Kbyte/sec for the VCR, while it is about 1330 Kbyte/sec for the machine on which we tested the CTR. Apart from this, CTR has not been designed to perform best for small messages in quiet networks. Small messages do not benefit from overlapped requests for instance, which results in relatively large delays. It remains to be seen how well the VCR performs with high loads on larger networks. What is more, the VCR needs large amounts of memory. Using default settings (160 bytes per buffer) the VCR kernel needs about 100 Kbyte plus 6400d bytes of buffer space per node, where d is the network diameter.

Tiny (Clarke and Wilson, 1991) is able to avoid deadlocks by eliminating cycles from routing tables. It allows routing without deadlock-avoidance as well. Clarke and Wilson present the travel times over 1 to 12 hops in a quiet network for messages of 4, 16, 64, and 256 bytes. They does not state whether these figures apply to deadlock-free routing or not, but this is probably not very important in this case. The raw link speed of the architecture used is about 770 Kbyte per second. For Tiny the same holds as for the VCR. It is fast for small messages in a quiet network, but it is not clear how efficient it is for high loads in a large network. It would be interesting to known if the elimination of cycles from routing tables affects the efficiency of Tiny in certain situations.

The adaptive router presented by Son and Paker (1991) has been tested on various network topologies, but the account of its performance is too short to draw any conclusions. It achieves a maximal average throughput of about 260 Kbyte/sec. The router cannot force packets to be delivered in the correct order, which can be a problem for some kinds of communication (I/O might need extensive buffering in certain situation). It does not necessarily route packets over the shortest path, in fact paths lengths may increase with network load. It is interesting to see that it performs best on a cross mesh, because this topology has many - but long - paths between any two processors. In contrast, the cross mesh is not exceptionally well-suited to the CTR, as the number of different shortest paths is small.

Except for the tests of the Parix system, all tests presented below were run on a Parsytec supercluster that contains 64 T800 transputers running at 25 MHz with 4 Mbyte of

5-cycle memory each (with error detection and correction). In this system all transputers are connected via a network of crossbar switches (Inmos C004 chips) that enable different network topologies to be used easily. The raw unidirectional link speed is about 1330 Kbyte/sec. For links that have to be set up via multiple cross-bars this number is even smaller. The tests do not make use of the on-chip memory of the transputer: one usually needs this for other purposes (see chapter 3). The initialisation code of the CTR takes about 4 Kbyte of code and the router itself about 8 Kbyte (excluding buffers and routing table). All tests of the CTR use adaptive routing with 10 zero class buffers per processor, except when stated otherwise. Additionally, a single buffer is allocated for each non-zero class. The number of classes equals the network diameter, even for topologies where less would be sufficient. The buffer size equals the maximum packet size. Thus, using buffers of 1 Kbyte and a network with diameter *d* the amount of buffer space per node equals *(10 + d)* Kbyte. All tests are simple loops that asynchronously transmit packets of a certain size to a particular destination, or to random destinations.

2.5.1. Helios versus CTR

In this section we will compare the performance of the CTR with that of the Helios (version 1.2) communication primitives. We will see that the CTR is significantly faster - especially for large networks - and that Helios is not entirely reliable.

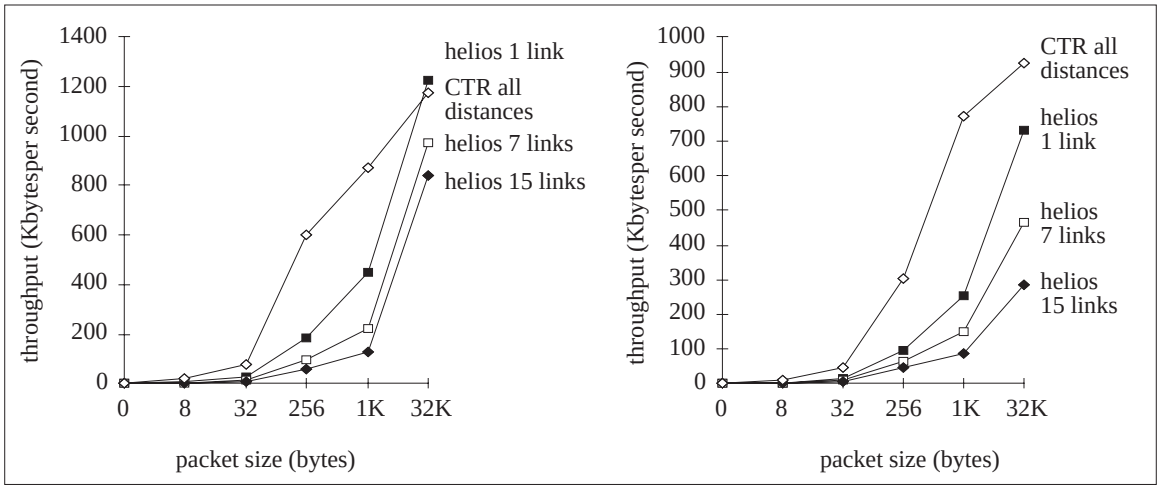


Figure 2-5: *Communication over an array of processors. The figure on the left shows unidirectional communications, whereas the other shows bi-directional communications.*

The figure above depicts performance measurements on an array of processors. Both CTR and Helios were tested with 2, 8 and 16 transputers. In the figure on the left all packets travel in the same direction from one end of the array to the other. In the figure on the right packets travel from one end of the array to the other end in both directions simultaneously. The picture shows the throughput in a single direction. The total throughput of the rightmost figure is twice this value. It can be seen that Helios slows down

when larger arrays of transputers are used. The CTR is not influenced by the network size in any way (which is actually what one would expect).

The following picture shows information about some tests on a grid and a ring. The figure on the left shows the throughput from one corner of the grid to the opposite one for both unidirectional and bi-directional communications between the two corners. The value for bi-directional communication indicates the throughput for one direction only: the total throughput is twice this value. The figure on the right shows the throughput from some source processor to random destinations (the number of bytes the source can send away randomly per second). This has been measured for a 2-dimensional grid of 16 processors and for a ring of 16. In case of the grid the source was either a corner processor or one in the centre. Helios does not take advantage of the network topology here. In contrast, the CTR is able to increase bandwidth by using multiple paths simultaneously.

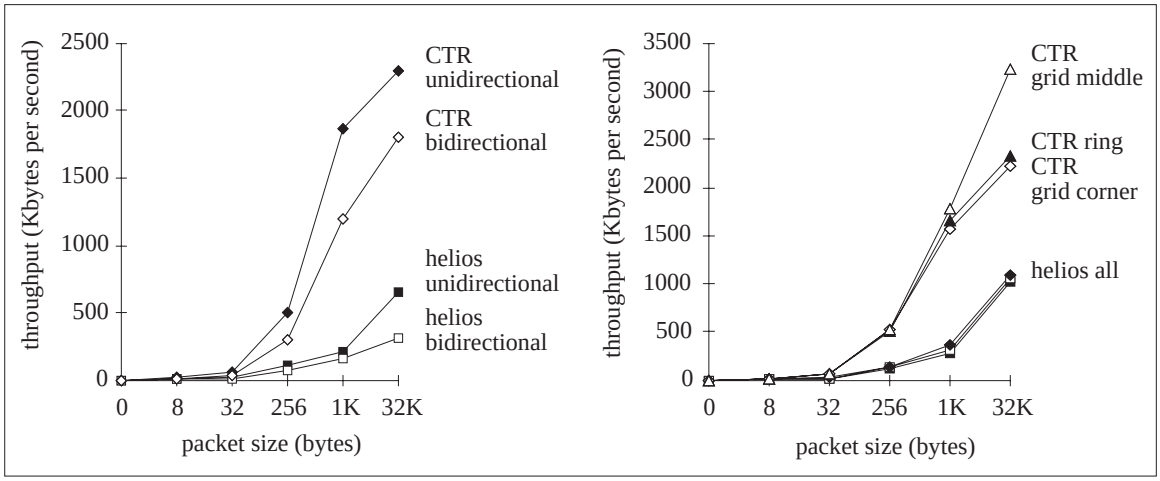


Figure 2-6: *Communication over a 2-dimensional grid of 16 processors and over a ring of 16 processors. The picture on the left shows communications between opposite corners of a grid. The picture on the right shows random communications from a processor on a grid and from a processor on a ring.*

The picture below shows the performance of the CTR and Helios when all processors are sending messages to random destinations, which means communications are frequent and chaotic. It shows the average throughput of a single processor. To obtain the total network throughput these values should be multiplied by 16: for the CTR using 1 Kbyte packets this is about 7 Mbyte per second on a grid, and about 5 Mbyte per second on a ring. Using a grid of 16 the average distance a packet travels is about 2.7, which means that about 19 Mbyte of data makes a hop per second, indicating every link transports approximately 790 Kbyte per second bidirectionally on average. For the ring the average distance is about 4.27 so that approximately 21 Mbyte traverses a link per second, resulting in an average bi-directional throughput of 1300 Kbyte per second per link.

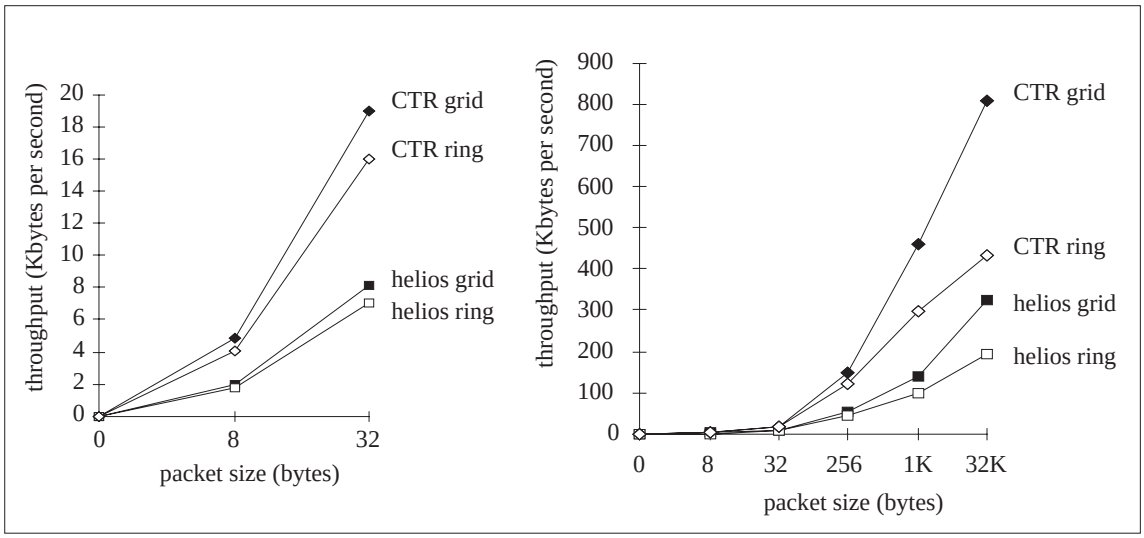


Figure 2-7: *Throughput per processor when all processors are sending messages to random destinations (i.e. the average number of bytes each processor can send away randomly per second). These results were obtained on a ring of 16 processors and on a 2-dimensional grid of 16 processors. The picture on the left is an enlarged part of the one on the right.*

In addition to the differences in speed between Helios and the CTR, Helios is not deadlock-free. Helios allocates routing buffers at runtime. This means that communications might fail when memory is low on some intermediate processor. According to the Helios manual messages may get lost due to congestion, which will be detected by an exception message that is returned to the sender. Unfortunately congestion may cause exception messages to get lost as well. As a result one needs time-outs on communication primitives, but there is no guarantee that subsequent tries will succeed. Tests have shown that programs that allocate a lot of memory at start-up may deadlock at the moment (heavy) communication takes place.

Looking at the results presented above it is clear that the CTR outperforms Helios. This may not be very problematic for common Helios applications, as the Helios programming philosophy is quite different from that of Concurrent Clean. Helios basically supports UNIX-style programs. For our purposes it is not very well suited. In the next section we will take a closer look the communication mechanism of Parix.

2.5.2. Parix versus CTR

We have been able to run some tests with the Parix communication mechanism. Unfortunately this had to be done on a different architecture as Parix was not available on our machine. In addition, porting the CTR to the Parix machine turned out to be a problem, due to the amount of work involved. Nonetheless, the following communication figures give some indication of the relative merits of both communication systems.

The Parix tests have been run on a network of 512 T805 transputers running at 30 MHz each having 4 Mbyte of memory. Only grid networks are possible in this system.

Using Parix one is able to achieve a maximum link bandwidth of about 1100 Kbyte/sec, but it is not clear whether this is the true hardware limit.

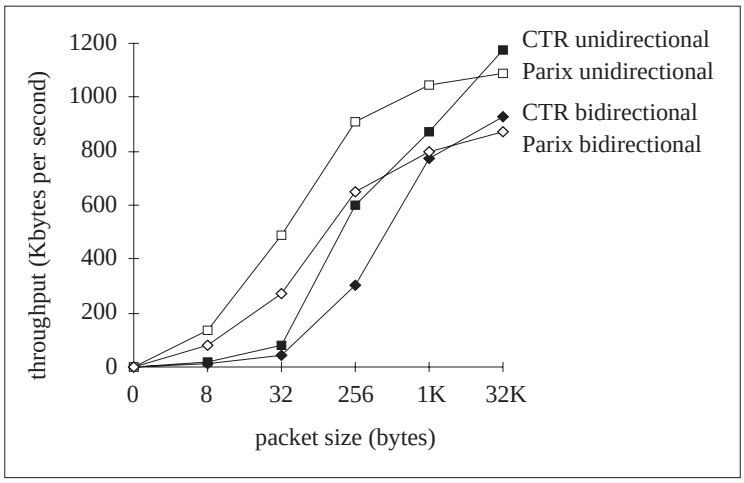


Figure 2-8: *Throughput over a single link in a quiet network. Parix is clearly faster than the CTR. Both unidirectional and bi-directional communication over a link has been measured.*

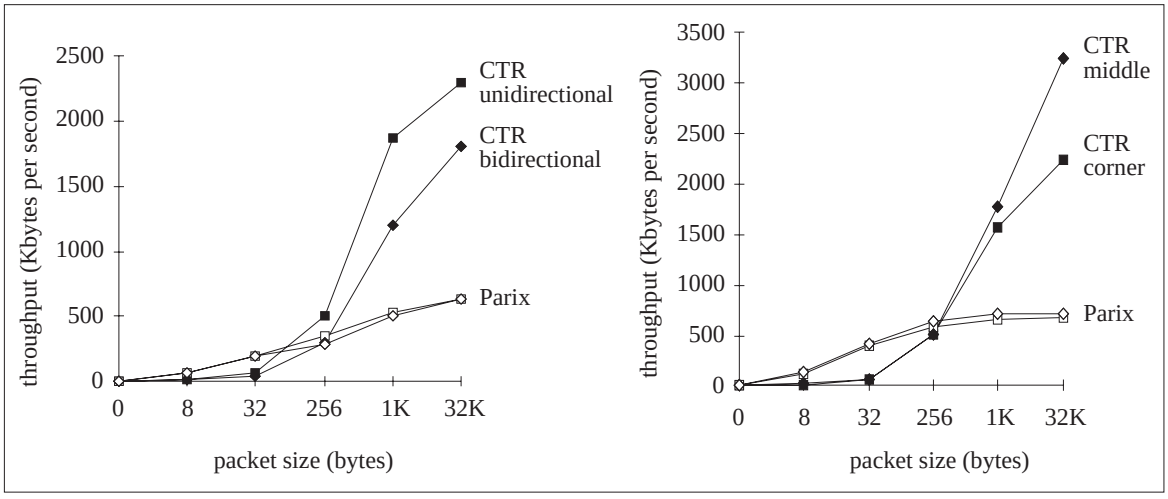


Figure 2-9: *Communication over a 2-dimensional grid of 16 processors. The figure on the left shows the throughput from one corner of the grid to the opposite one for both unidirectional and bi-directional communications between the two corners. The value for bi-directional communication indicates the throughput for one direction only: the total throughput is twice this value. The figure on the right shows the throughput from some source processor to random destinations (the number of bytes the source can send away randomly per second). The source was either a corner processor or one in the centre.*

Parix is based on the use of virtual links. These have to be allocated at runtime. It is not clear how much memory each virtual link takes. As can be seen in the pictures above,

using grids, Parix is very fast for small messages, about as fast as the CTR for messages around 256 bytes, and rather slow for messages exceeding 256 bytes. On the one hand Parix is faster than the CTR for communication over a single link in a quiet network (see figure 2-8), but on the other hand, performance of Parix decreases with increasing distance (figure 2-9). We were planning to test how well Parix performs when all processors send randomly destined messages, but unfortunately we did not succeed in setting up a fully connected network of virtual links over a grid of 16 transputers, due to limitations of Parix.

Concluding, it depends on the communication pattern needed which kind of routing mechanism should be used. For the tests presented above Parix is clearly best when small messages have to be routed. The CTR may be better for large messages. As we have not been able to test Parix when all processors send messages to random destinations it is not yet clear how Parix compares with the CTR in this respect. In the next subsection, we will present some experiments with various parameter settings within the CTR.

2.5.3. The influence of packet storage size

The number of zero class buffers per processor influences the performance of the router. The more buffers are available the smaller the chance that requests need to be cancelled. When very few buffers are available it is difficult to use multiple links simultaneously.

The figures below present some tests with packets of 256 bytes on a few simple topologies. We used arrays of transputers and avoided tiny messages to ensure that data traffic was high enough for all links. This reduces protocol overheads due to other causes than lack of buffers. On the other hand, we have not used very large messages, as these tend to decrease all protocol overheads. These tests - and others on different topologies - have indicated that about 10 zero-class buffers are sufficient in general. Adding more buffers will sometimes increase performance slightly for larger topologies, but we have not yet encountered an example where this resulted in a significant speed-up.

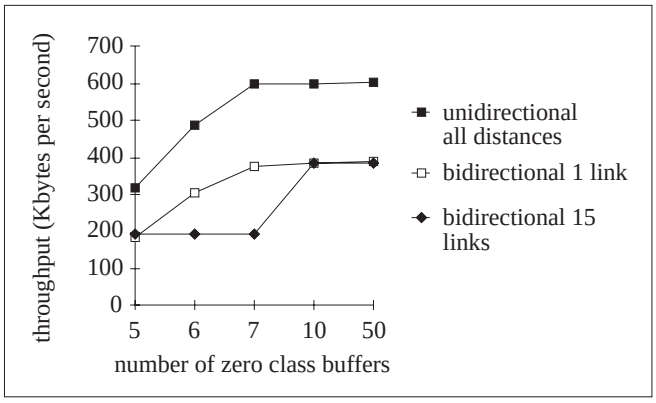


Figure 2-10: *Performance on arrays of various sizes for different numbers of zero class buffers (with packets of 256 bytes). For reasons of efficiency the current implementation requires the number of zero class buffers to exceed the number of transputer links (4).*

2.5.4. Adaptive routing versus fixed routing

For point-to-point communications the CTR allows to use either an adaptive or a fixed strategy. The former routes packets over an arbitrary shortest path and does not guarantee packets to be delivered in the same order they were sent. In contrast, the fixed strategy uses a fixed shortest path to route packets, which keeps packets in the right order. The ability to keep packets ordered can be very useful for some kinds of communication (file IO for instance). One-to-all broadcasting is supported as well, but for this no performance figures are available yet (the Clean runtime system does not yet take advantage of broadcasting, except for loading program code).

Depending on topology and network load, adaptive routing is able to improve performance. It does introduce some additional overhead though, so that it will not always be better to use adaptive routing. The figures below show that adaptive routing is best for large packets and high loads. Additional tests have shown that the effect of adaptive routing diminishes when the input load is decreased. When packets are small or the network is not very busy - which implies that relatively much control information is needed - the link speed is not the limiting factor and a single path is able to do the job just as well as multiple paths, sometimes even better as a single reasonably well-used path introduces less useless control messages (signals followed by cancels) than several barely used paths.

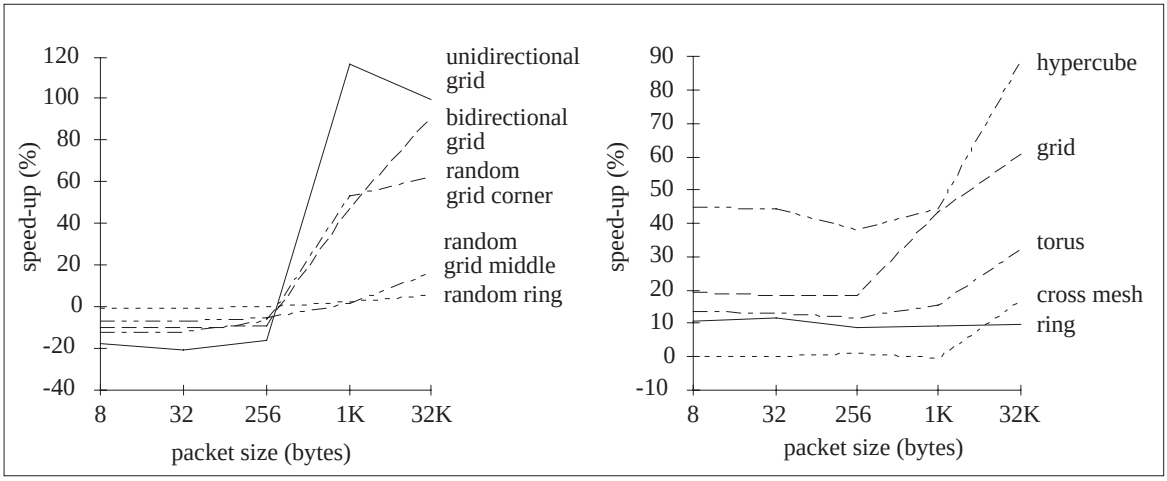


Figure 2-11: *Speedups that have been obtained by using adaptive routing instead of fixed routing. The following tests are shown in the figure on the left: unidirectional communication between opposite corners on a grid of 16, bi-directional communication between opposite corners on a grid of 16, randomly destined packets from a single processor on a ring of 16, randomly destined packets from a corner processor on a grid of 16, and randomly destined packets from a 'middle' processor on a grid of 16. The figure on the right pictures tests in which all processors simultaneously send randomly destined messages. This has been done for the following topologies: ring, cross mesh, torus, 2D grid, and hypercube.*

The last observation also explains the slight 'dip' for some topologies in figure 2-11. This temporary decline in efficiency for adaptive routing occurs for packet sizes around

256 bytes. At this point, the protocol overheads decrease for fixed routing because paths become more heavily loaded. This point arrives later for adaptive routing, so that the relative efficiency of adaptive routing drops temporarily.

2.5.5. The influence of different topologies

The next pictures show the throughput for different topologies when all processors send messages to random destinations. All measurements have been performed on networks of 16 transputers. They indicate the number of bytes a single processor was able to send away per second. For the total network throughput this figure should be multiplied with 16. As can be seen in this figures, the CTR is able to take advantage of network topology. The cube and torus perform best for networks up to 16 processors. For larger networks it can be expected that a cube (diameter of $O(log(n))$) performs better than a torus (diameter of $O(\sqrt{n})$). Unfortunately, it is impossible to construct larger cubes with the transputer hardware, as each processor can be connected to at most 4 others.

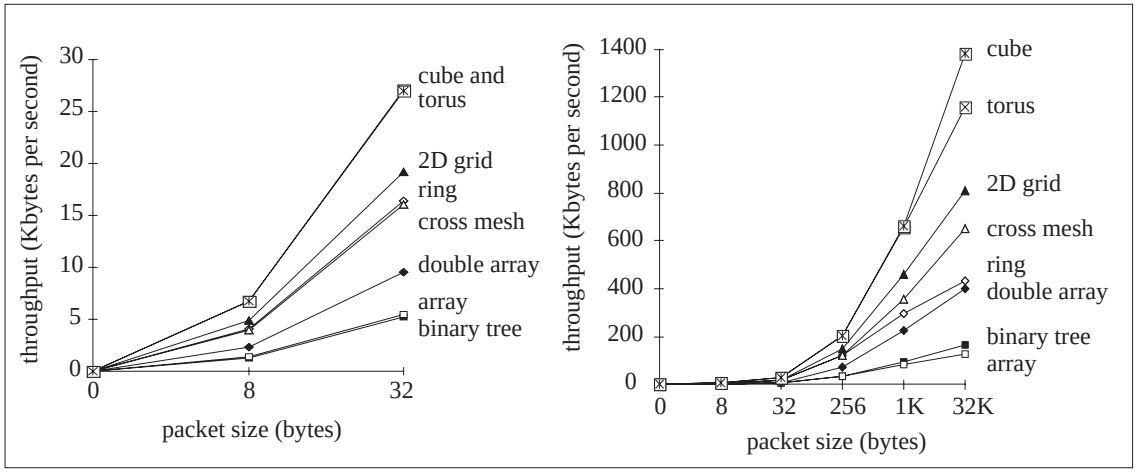


Figure 2-12: *Throughput for various topologies using adaptive routing. The picture on the left is an enlargement of the one on the right. All networks consist of 16 transputers. Most topologies are self-explanatory. The double array is an array of which all connections consist of 2 transputer links instead of 1. The cross mesh is depicted in figure 2-13.*

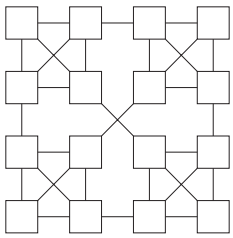


Figure 2-13: *The cross mesh*

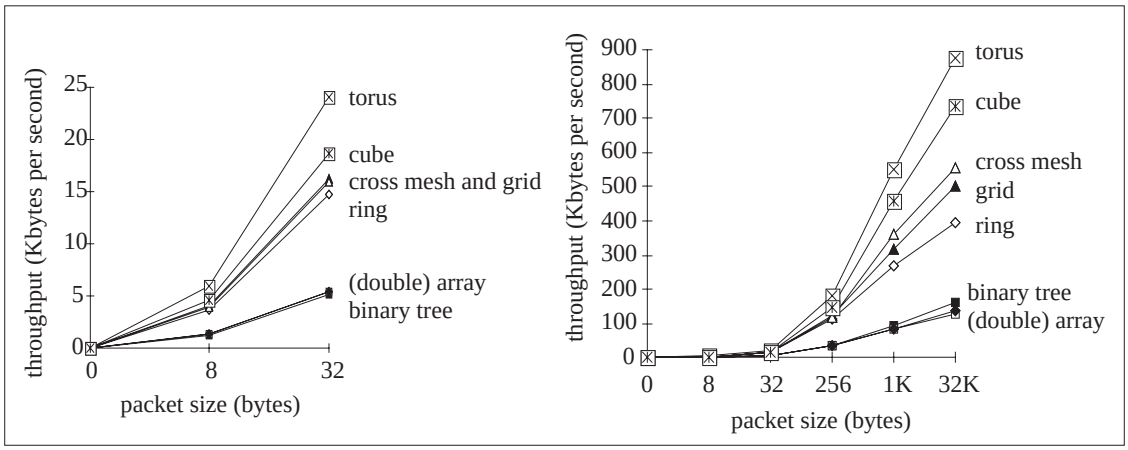


Figure 2-14: *Throughput for various topologies using fixed routing. The picture on the left is an enlargement of the one on the right. All networks consist of 16 transputers. See figure 2-12 and figure 2-13 for an explanation of the topologies used.*

2.6. Conclusions

Eventually, when choosing a communication mechanism, a main question will be whether large messages are very common in Concurrent Clean programs. On the one hand the Concurrent Clean graph copying mechanism tries to construct messages that are as large as possible, as it *assumes* that large messages are relatively cheap. On the other hand, some programs simply do not start up processes on large graphs. Nonetheless, we should take into account that communication of large graphs can seriously influence performance *if* such large graphs exist. In addition, a system that is relatively cheap for large messages makes it possible to improve performance by combining messages, which is a clear concept from a programmers point of view.

In this chapter we have presented the Class Transputer Router, an adaptive router that uses class climbing to avoid deadlock. This router is based on the algorithm that was used for the routing chip of the DOOM architecture. We have given a brief description of the original algorithm and the changes that were needed to adapt it to the transputer hardware. Performance figures indicate the router is able to compete with the Parix communication mechanism and is faster than the Helios communication primitives. The required communication pattern determines largely which kind of routing mechanism is best. In general the Class Transputer Router performs very well for high network loads, taking advantage of network topology.

Thus, we have given each transputer the capability to communicate with any other processor in a network of arbitrary complexity. This enhancement provides a general platform of a higher level that is compatible with architectures that incorporate a hardware routing mechanism, like Thinking Machines' connection machine, and with systems that use some software solution to provide message passing, such as networks of SUN's running the PVM package. The next chapters will show how we have implemented Concurrent Clean on it.

3. Realising the Components of the Abstract ABC Machine

The previous chapter has presented an efficient way to realise reliable communications between processors, and therefore, between processes. However, we do not yet have any processes that can transmit messages, nor any data structures that can be communicated.

The Clean compiler compiles a (parallel) Clean program to code for an abstract machine, the (parallel) ABC machine (see section 1.5.6). In this chapter we will present - and explain- the concrete realisation of the most prominent components of the abstract ABC machine: the stacks, the heap, the processes, and the nodes. Designing these elements requires careful consideration of their (intended) use. This is closely related to the research issues that have been raised in section 1.3 and 1.4. Amongst others, one needs to consider the workings of the concrete machine architecture (i.e. the transputer hardware, see section 1.6), and the decision to make no concessions with respect to the generality of the implementation. Furthermore, the aim to base our implementation on sequential compiler technology needs to be taken into account. Many additional issues are important, and therefore this chapter will start with a review of the most significant ones.

In section 3.1 we will recapitulate some basic research issues and relate them to the design of the parallel ABC machine. Section 3.2 will expand on the consequences of one of these issues, namely the decision to allow speculative parallelism. Sections 3.3 to 3.6 will present the concrete realisation of respectively the stacks, the heap, the processes and the nodes.

3.1. The basic research issues and the design of the ABC machine

This chapter revolves around basic design decisions. It explains how our views culminated in concrete organisations on parallel hardware. Our goal is to get an efficient implementation of a lazy functional language on a concrete general purpose parallel machine. The decision to focus on general purpose machines, as opposed to developing a dedicated processor has mainly been based on the expectation that the latter would eventually result in a relatively slow machine, as industry continues to put massive efforts into development of the former. This view has turned out to be correct considering the generally rather disappointing results that have been obtained with specialised hardware so far (Darlington and Reeve, 1981; Watson and Watson, 1987; Peyton Jones *et al.*, 1987;

Hankin *et al*., 1985; Richards, 1985; Stoye, 1985; Keller *et al*., 1984; Anderson *et al*., 1987; Magó and Stanat, 1989). Consequently, research on special purpose hardware has largely been discontinued.

Furthermore, we have restricted ourselves to architectures with distributed memory. These are more scalable than shared memory machines, for which memory access constitutes a bottleneck. Typically, shared memory architectures contain no more than a few dozen processors, while machines with distributed memory contain anything between one and a few hundred, and - if one can afford it - thousands.

We concentrate on true distributed memory systems. Hybrid designs that combine distributed memory with hardware to present it as shared memory fall outside the scope of this thesis. They behave like distributed memory machines in the sense that access to remote memory is more expensive than local access, but provide the easier shared memory programming model. As to caching mechanisms employed in these virtual shared memory systems, it is interesting to note the similarities with graph copying. Both are employed to provide efficient data access. Although we do not treat caching explicitly, we do use graph copying as a caching mechanism. In a way, one may think of Concurrent Clean as a virtual shared memory implementation.

The concepts of the parallel ABC machine directly originate from the decision to focus on general purpose architectures with distributed memory. Dealing with distributed memory, we assume that communication is expensive compared to execution of code. In particular delays will be high (see also chapter 2). This has a number of consequences. First of all, we primarily support coarse grained parallel processing, so that communications are suppressed and relatively well-known sequential compiler optimisations can be exploited.

At the same time, we allow a considerable number of processes to be running interleaved on a single processor. The exact number is determined by the availability of memory only. Having many processes, the effects of communication delays can be hidden to some extent, as increasing the number of processes decreases the probability that a processor runs out of work and has to wait. On the other hand having more processes means more process management is needed and clearly, there is a trade-off between the size of processes and their number. So, although in principle we promote coarse-grained parallelism, it is important to support finer grains as much as possible when pseudo-parallelism on a single processor is involved. This means that lightweight processes should be provided.

And finally we have to keep communications - expensive as they are - as lean and efficient as possible. This involves avoiding small messages, which are relatively costly, and allowing speculative parallelism (see section 3.2), so that there is a possibility to compute - and transmit - results in advance without requiring strictness. The latter is related to buffering and introduces some slackness in data dependencies. It may avoid communication delays in acquiring results and is mostly useful for asynchronous stream processing (we will see some more uses later). This is the main difference with implementations like Wybert (Langendoen, 1993) and ZAPP (Goldsmith *et al.* 1993), that concentrate on divide-and-conquer parallelism only. All this has a considerable impact on the structure of the final system.

Sequential compiler technology forms the backbone of our parallel implementation. It will become clear that it is very well possible to combine structures for sequential processing with those for parallelism, although it requires some careful designing. Sequential implementations typically employ a set of stacks and a heap to perform graph rewriting. Carrying this concept to a parallel world, we need stacks for every process and a heap that can be shared by all processes on a single processor (sharing avoids communications). On a lower level, registers are needed for each process, so that important optimisations can be employed. These provisions for efficient sequential processing should not cause serious overheads in parallel processing, in particular with respect to process management.

Conversely, parallel constructs must be efficient and should not hamper sequential processing. For example, our objective to exploit speculative parallelism requires the incorporation of a fair scheduler, but this should not lead to high context switching overheads (see the following section). If feasible, parallel constructs must be designed so that they do not rule out code optimisations nor increase memory requirements. The next sections will demonstrate how to devise structures for effectively combining sequentialism and parallelism.

3.2. Dealing with speculative parallelism

The decision to allow speculative parallelism has a serious impact on the way an implementation should be realised, and thus, on the design of the parallel ABC machine. In general, speculative parallelism complicates matters, and therefore many functional languages do not allow it. We will elaborate this next.

3.2.1. The costs of speculative parallelism

Some have contended that managing speculative parallelism will inevitably lead to high runtime overheads, and that it will be extremely difficult to incorporate it in a parallel implementation (Peyton Jones, 1989-c). Based on such claims many have refrained from considering the implementation of this feature. However, reviewing the arguments against speculative parallelism we have come to a different conclusion.

Avoiding speculation has been based on two assumptions. First of all, avoiding termination problems is considered to be very expensive. And secondly, it has been argued that it is very hard to remove speculative tasks that have turned out to be irrelevant. We will show below that these assumptions are not always valid.

To begin with, termination problems can be avoided in two ways. One can either use fair scheduling, or priority scheduling. Both are generally considered to be expensive. This may be true for many architectures, but on the transputer hardware, fair scheduling can be implemented very efficiently. On a transputer, context switches are barely noticeable (a context switch takes less than 1 μs., while a time slice period takes between 1 and 2 ms.). It is conceivable that similar features will be incorporated in future general purpose hardware designs, as pre-emptive scheduling of processes is becoming rather common. We will treat this in more detail later.

And secondly, removing irrelevant tasks is *not always* hard. In Concurrent Clean, speculative parallelism will commonly be employed to achieve some kind of buffering behaviour to avoid delays. Typically, speculative tasks will not need to run indefinitely, but for a bounded amount of time (or space). This can be enforced by giving them a limited amount of resources, so that they cannot consume too much of it. The exact amount is determined by the delays of the communication hardware (which means that a program needs to be tuned). Employing such a strategy, speculative tasks will stop themselves when they run out of resources, so that no special external mechanism is required to kill them. We will see some examples of this in chapters 7 and 8 where we use annotations and skeletons to accomplish this buffering behaviour. So, in these cases one does not have the problem of processes that actively produce garbage, but the convenience of processes actively killing themselves. As a result, it will not always be necessary to use an additional - and possibly expensive - runtime garbage collection mechanism to detect irrelevant processes.

The only processes that constitute a problem are irrelevant speculative ones that do not fit in such a scheme and keep running too long (or indefinitely). Unfortunately, one cannot preclude such unbounded speculative processing if a programmer is allowed to eagerly start processes by means of annotations. We will see in chapter 5 that it may be possible to develop effective garbage collection methods that deal with this problem as well. It still remains unclear whether these methods will be truly successful. Until then, it is too early to conclude that speculative parallelism is too difficult to handle.

On the other hand, we expect that unbounded speculative processing will be relatively rare. Unbounded speculative processing is hardly useful, and will normally not be introduced because of the risk of wasting machine resources. One might indeed question whether it is useful to support unbounded speculative parallelism at all.

3.2.2. Avoiding termination problems: fair scheduling or priorities.

If we allow (bounded) speculative processing we must determine how to implement it. The main problem is how to keep non-terminating speculative processes from stopping the whole computation. Basically, there are two techniques: fair scheduling and priority scheduling. Fair scheduling avoids termination problems by ensuring that *all* processes eventually get some processing time. In contrast, priority scheduling assumes that the execution of high priority processes prevails over the execution of low priority ones, so that low priority processes will be interrupted by processes of higher priority. It avoids termination problems by giving speculative processes a lower priority than non-speculative (conservative) ones.

The advantages of priority scheduling are that speculative parallelism can be incorporated safely without risking serious context switching overheads for conservative tasks. Needed computations can simply proceed without interruption. If we merely consider an evaluate-and-die model of processing where processes reduce to root normal form only, and if we assume that every process runs for approximately the same amount of time, this has the additional advantage that needed results will probably be delivered earlier than they would have been using fair scheduling.

However, the advantages of priority scheduling depend heavily on the assumption that context switches are expensive. As we have above, this is not true for the transputer hardware. In addition, priority scheduling does have some disadvantages.

- First of all, priority scheduling sometimes causes expensive priority updates. A speculative task may become needed and then its priority should be increased. The priority of some of its child tasks might have to be increased as well, and so on. This can be costly.
- Secondly, priority scheduling can be a problem in lazy languages that use strictness analysis to discover needed computations. Unfortunately, strictness analysis can only *approximate* actual strictness properties. This means that some computations are assumed to be speculative, while in reality, they are not. Running such computations at low priority will certainly introduce priority upgrades, so they should have been run at high priority in the first place (however, one might argue that in such cases, the programmer should have indicated strictness).
- Thirdly, some forms of processing require *concurrent* evaluation of (partial) results. For example, producers and consumers of streams often should not be allowed to run until completion. A producer should not compute a complete stream before a consumer is allowed to consume it. Similar problems arise when concurrent I/O is involved or when one process takes considerably more time to complete than others (that follow). A fair scheduling mechanism will then be needed to enforce concurrent evaluation of all results.
- And finally, having low priority processes for speculative computations can increase delays. In a distributed system, speculative computations are needed to avoid delays, and consequently they should not be computed too late. A consuming process might even request the result of a speculative computation earlier than some strict results, as (the absence of) strictness does not say anything about *when* a process needs a particular result. In other words, (bounded) speculative parallelism addresses the problem of lazy evaluation in a distributed environment. It provides a way to start computing results *before* they are needed.

Consequently, Concurrent Clean uses fair scheduling for its implementation. We argue that not only the evaluation of irrelevant speculative tasks wastes machine resources, but also delaying the execution of relevant ones. One does not know in advance which speculative tasks are useful, so it is incorrect to assume they are less - or more - important than other tasks. However, one might trust the programmer - or some analysis technique - to introduce speculative tasks only if there is reason to do so, that is, if it is likely they will be useful. The whole point of introducing speculative processes, is that they will use more resources. This sharply contrasts with the view of Peyton Jones (1989-c).

Note that this discussion also reveals a problem of task pools that hold processes that might be scheduled when some processor runs idle, which is useful to avoid flooding a system with running processes. One should make a distinction between annotations that indicate functions that are *eligible* for parallel evaluation (based on computational complexity and communication overheads) and functions that *should* be run in parallel to obtain the intended parallel behaviour. In a distributed system it is important to start up some processes early to avoid delays. Additionally, concurrent I/O requires certain

processes to be started up concurrently. For this reason the Concurrent Clean system starts up processes as soon as an annotation is encountered. We have not yet considered task pools. Annotations can be used to avoid flooding.

3.3. Registers and stacks

Traditionally, optimising compilers largely depend on the availability of registers to cache frequently used values. Taking into account the importance of registers, which is confirmed by their widespread presence in modern processors, the ABC machine has been designed to take advantage of them. If an architecture has a reasonable number of general purpose registers the ABC machine can conveniently be implemented on it.

Sequential implementations of the ABC machine not only employ registers to evaluate expressions quickly, but also for providing fast access to important structures such as the heap and the stacks. This means that at least three registers are needed. One to hold the heap pointer and two for the stacks, as the b- and c-stack are normally combined. The stacks themselves are used as an extension of the limited amount of registers (which automatically makes registers an optimisation for stacks, depending on your point of view). Taking advantage of stacks and registers, as opposed to using graph rewriting in the heap, has been known to give a considerable performance boost for certain functional programs.

Unfortunately the transputer does not have sufficient general purpose registers to implement the parallel ABC machine in the same manner. This has forced us to find a different way to establish efficient access to the stacks and the heap, which is crucial for good performance. We will take a short look at three alternatives next.

3.3.1. Workspace for stack

Several transputer programming languages use the transputer workspace pointer as a stack pointer. This is straightforward if a single stack suffices, which is the case for languages like Occam and C. Pointers to global data are carried around on this stack. C implementations for instance, pass pointers to global data as implicit function arguments. For the PABC machine adoption of this solution would lead to problems.

Following the example above, one could represent one of the PABC stacks by the workspace pointer directly. Pointers to the other structures - the other stack pointer and the heap pointer - would have to be carried around on this stack, resulting in an extra indirection. Unfortunately these pointers are used much differently than the global data pointer found in C implementations. First of all, the heap pointer is not a constant, so it not only needs to be passed to each function, but it must be returned as well. Updating the heap pointer becomes expensive this way. Depending on the handling of stacks, in particular on stack overflows, a similar situation exists for stack pointers. Furthermore, a functional program usually accesses the stacks far more frequently than imperative programs use the global data pointer. As a rule, important data consists of local variables and function arguments, and these are supposed to be on the stack. If they are not, C programmers - and sometimes the compilers - typically use the stack to cache frequently accessed global items or references. They create an alias. This trick sometimes works for imperative languages, but not for the ABC machine. Firstly, because it does not always use a few stack elements

frequently, but rather many elements a little. And secondly, because stack elements of one stack cannot simply be stored on the other for a long period of time, due to garbage collection. As a result, the extra indirections for addressing stack elements impose too much overhead for functional languages.

3.3.2. Merged stacks

Ignoring the implementation of heap allocation for the moment, the extra indirections to access stack elements can be avoided if all stacks are merged into one. In other words, by caching the entries of one stack on the other one for an indefinite period of time. A slight drawback of this solution would be that elements may have to be rearranged more often than is the case for separate stacks, but the main advantage is clear: the transputer workspace pointer can be used to refer to the top of the single merged stack.

However, the garbage collector should still be able to distinguish the pointers from the non-pointers on the stack, which was the reason for having multiple stacks in the first place. As a result, stack elements have to be tagged in some way. Many solutions need special distinct representations for different kinds of stack elements, or vectors that are inserted between ordinary stack elements. These imply notable computational and spatial overheads. A solution that does not suffer from these problems is the one employed by the partial implementation of Concurrent Clean on ZAPP. It has a merged stack and avoids explicit manipulation of the stack to store context information by associating this information with each return address on the stack. Return addresses enclose sets of stack entries and for each set the return address on top uniquely identifies which entries are pointers and which are not. This imposes no overhead on normal execution and complicates garbage collection only little.

Although this appears reasonably satisfactory, we have chosen not to use this technique. The heap pointer still has to be accommodated in a different way, and apart from this, there are important reasons to keep the transputer implementation in accordance with the implementations for register based machines as much as possible. We will discuss this in the next section.

3.3.3. Virtual registers

We have chosen to deviate as little as possible from implementations for concrete register-based machines. There are various reasons for this. First of all, it is unclear whether using the workspace pointer as a stack pointer is truly advantageous. Secondly, the transputer architecture is not very common and in particular the T800 is quite old. Most modern general purpose processors do have a reasonable number of registers and there is no indication this will change in the near future. On the contrary: newer processors rather contain more registers than older ones. In a way, this holds for newer transputers as well. The T9000 transputer incorporates instruction grouping and caches that allow the workspace of a process to be used as a set of registers without serious loss of efficiency. We were reluctant to develop new implementation techniques for an old architecture. And finally, we wanted to investigate which existing optimisation techniques for register-based sequential implementations can be applied to register-based parallel ones as well. Not only do we aim to keep our parallel implementation compatible with the sequential

implementation of Clean for Macintoshes, Suns and PC's, but also to keep it related to common (sequential) implementations of other languages. This could indicate future directions for research on code optimisation, given the continuing importance of sequential architectures and the growing significance of parallel ones.

This meant we had to provide registers in software to compensate for the lack of hardware registers, without killing performance. For this we used the on-chip static RAM of the transputer. Access to on-chip memory is considerably faster than accessing normal RAM - up to a factor of five. We have used this on-chip memory to give each process a kind of private register set that is used in exactly the same way as a real register set in sequential implementations: for evaluation of expressions and to accommodate stack pointers and heap pointers (for more information, see the process structure in figure 3-2). As will become clear, many optimisations remained applicable and this turned out surprisingly well, despite some extra - but cheap - indirections needed for addressing. In general, our compiler generates more efficient code than other transputer implementations, including the one on ZAPP.

To stress that we are not restricting ourselves to transputer hardware we prefer the familiar term 'registers' for this emulated register set. This conveniently relates our implementation to more common architectures. To avoid confusion with the hardware registers of the transputer we will sometimes talk about the 'pseudo-registers', or 'emulated registers'. The transputer registers will be referred to as 'evaluation stack', 'transputer stack', or plainly 'transputer registers'.

3.4. The heap

During graph reduction numerous nodes are created and this requires fast heap allocation and reclamation. In Clean, heap objects can have various sizes. This makes the use of free-lists less suitable to administer free memory. A better - and well-known - way to provide fast allocation in this situation is by using a compacting garbage collector that ensures the free space occupies a contiguous piece of memory. A register can then be allocated to point to the first free byte in memory. To reserve space one only needs to take this pointer and advance it the required number of bytes. This method has been successfully employed in many sequential implementations.

It is not entirely trivial to take this concept to parallel implementations, because all processes - both low priority and high priority ones - should be able to allocate memory from the same heap in an indivisible way. Processes must now share the pointer to the free area and they must be able to update it in an indivisible way. This is a delicate problem on the transputer, due to its automatic time-slicing features and the implementation of its two priority levels. Locking is costly (see the frame 'Atomicity on the Transputer') and the heap pointer cannot plainly be kept in an emulated register, because the emulated registers are not physically shared by processes (to keep context switches fast). On other architectures similar problems may exist as well.

Atomicity on the Transputer

Preservation of atomicity on a transputer requires careful consideration of its scheduling mechanisms. It is easy to support atomic actions among processes of the same priority. High priority processes are not time-sliced at all and there are only a few instructions that may cause descheduling of a low priority process. Avoiding these instructions within a sequence of code will ensure indivisibility within the scope of low priority processes. It is more difficult however, to ensure indivisibility among processes running at different priority levels. High priority processes may interrupt low priority processes at any moment - which is the exact reason for having them -, and this means that low priority processes need to increase priority temporarily if shared data is accessed.

Unfortunately, changing priority is relatively expensive on the transputer. It involves starting up a new process on a different priority level, while stopping the current one. This is illustrated below. While this is appropriate for an implementation of semaphores in general, it is not suited to efficiently provide tiny atomic actions, such as heap allocation.

increase priority code	cycles	
`ldc proceed-2`	1	
`ldpi`	2	
`stl -1`	2	*Store the address of the label 'proceed' at wsp-1.*
`mint`	1	
`ldnlp dummy_wsp`	1	
`gajw`	2	*Switch to a dummy workspace.*
`runp`	10	*Run a new high priority process with the old wsp. It will immediately interrupt the current process and start at the label 'proceed'.*
`stopp`	11	*The interrupted low priority process is still active. It will stop itself later and store its instruction pointer in the dummy workspace.*
`proceed:`		

decrease priority code	cycles	
`ldlp 0`	1	
`adc 1`	1	
`runp`	10	*Start a new low priority process and stop the current high priority one. The 'stopp' instruction will store the start address for the low priority process at the correct place.*
`stopp`	11	

3.4.1. Hierarchical heap

We solved these problems by giving each low priority process a small individual heap - a few kilobytes - from which it allocates memory. Every process keeps a private pointer to the first free byte in its little heap, so it can quickly - and safely - reserve memory in exactly the same way as in sequential implementations. If it runs out of local heap space, it calls the garbage collector to free unused memory.

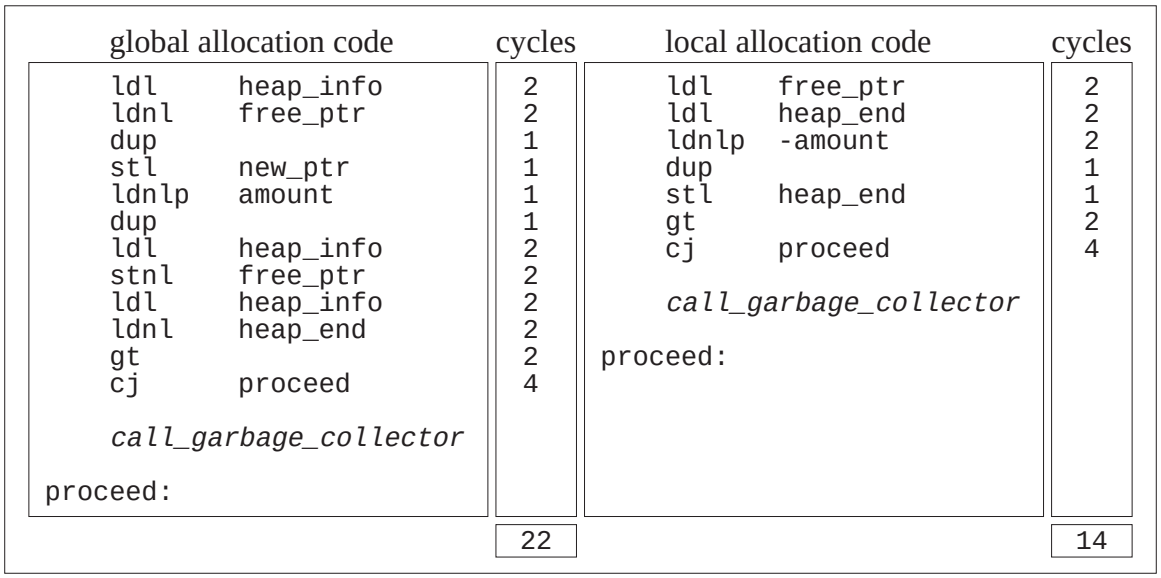


Figure 3-1: *Local versus shared heap allocation costs.*

The little process heaps are located inside the main processor heap, carefully kept invisible to all but the garbage collector and the high priority (system) processes. Usually the garbage collecting process will merely reserve a new heap without actually removing garbage. Only when the main heap becomes exhausted it starts recovering unused space. This means we have shifted the problem of performing safe memory allocations to a single high priority process. Semaphores are still needed in some form to invoke it, and although this is relatively costly compared to forwarding a pointer there is no serious loss in efficiency, because it occurs infrequently. On the contrary, in figure 3-1 we can observe that maintaining a local heap pointer instead of a global one gives considerable performance gains. The many cheap allocations in the local heap easily outweigh the few expensive ones in the global heap.

For comparison, the ZAPP implementation and the HDG machine have avoided the costs of locking on the transputer in another way. They both unshare part of the heap administration by allocating from one end of the heap at low priority and from the other end at high priority. This works correctly if low priority processes do not allocate more than a certain amount in one go and high priority processes ensure they leave this amount free. However, the heap pointers are not stored locally in registers, which makes this solution less efficient than using private heaps. In addition, it cannot be applied on hardware with more than two priority levels.

Some concluding remarks need to be made about having the hierarchical heap presented above. First of all, this concept relates to generational garbage collection, which is based on the observation that heap objects usually have a short life-cycle. The private heap of each process seems a natural construct to provide generational garbage collection on a per process basis. This not only can avoid traversing a lot of old objects during garbage collections, but it also improves locality of heap usage. Secondly, at some level it may be important to have a way to control the speed at which a specific process runs. It is in particular useful to keep a process from flooding the main heap which may impede other

processes. The private heap of a process provides a handle that can be used to monitor and restrict a single process, for example by withholding a new private heap for some time if it allocates memory too quickly. And finally, as we will see later on, having localised registers, stacks, and the heap to a single process it becomes possible to improve the caching behaviour of a parallel system, provided the right hardware is available.

3.5. Processes

At the lowest level, each process is implemented as a transputer process that is scheduled automatically in a fair way. At a higher level, it can be viewed as a set of registers, stacks, and a private heap. We have depicted the relation between all components in figure 3-2 below.

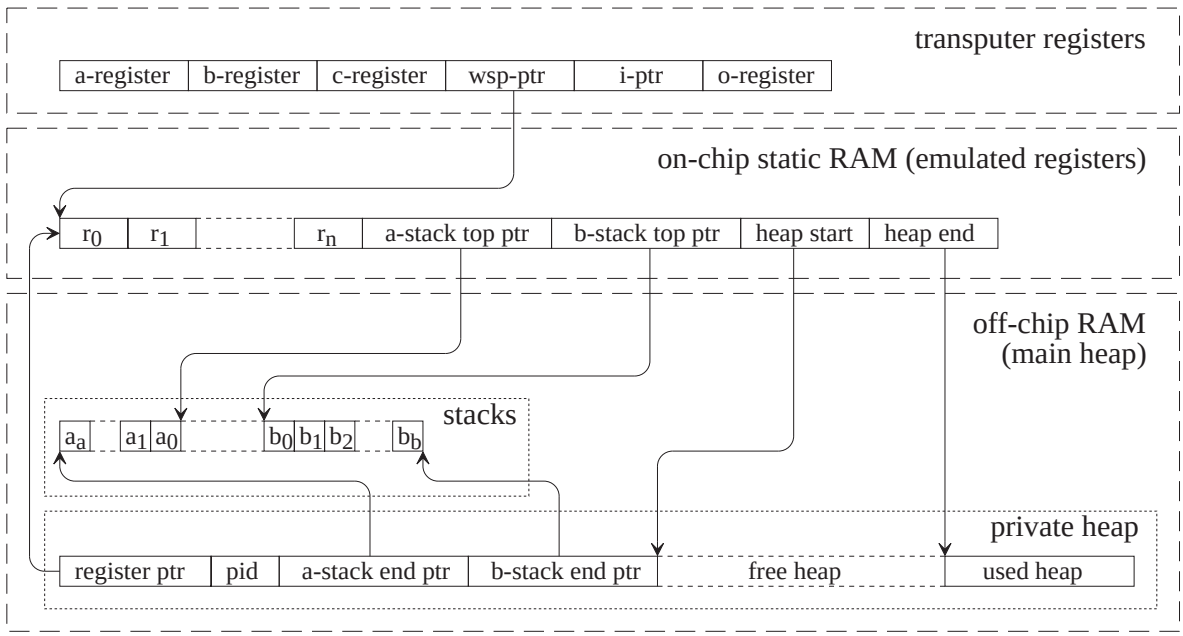


Figure 3-2: *Process structure. The stacks and the private heap of each process are located in the main heap.*

This sharply contrasts with the Wybert implementation (Langendoen 1993). Wybert does not incorporate fair scheduling and it does not have separate stacks for processes. Instead, all processes execute on the same stack. Only the processes on the top of the stack may proceed. This means it does not have concurrently executing processes on a single processor. These differences are caused by different goals: Wybert is designed for divide-and-conquer parallelism only.

3.5.1. Virtual virtual registers

The amount of on-chip memory on a transputer is limited and so is the number of register sets that can be located on it. The current size of each register set (16 words of 4 bytes each) allows about 50 processes to be accommodated in on-chip memory. Such a limit is

not acceptable. Language semantics dictate that we need to be able to run an arbitrary number of processes on a single processor. To solve this we have made our registers more virtual than they already are.

We have employed a mechanism similar to a paging mechanism in virtual memory management to give each processor and unlimited number of register sets. In contrast to what the picture above suggests, registers are actually not located in internal memory, but in the heap. In addition to the transputer scheduling mechanism we have implemented a process manager that ensures the registers of a process get located in internal memory by the time the transputer hardware runs it. After consuming a time-slice the registers get swapped out to external memory again. Processes do not notice this. It appears to them they are always located in internal memory, albeit at varying addresses.

To accomplish this we allow the hardware scheduling list of the transputer to contain merely on-chip processes and the process manager. The latter gets scheduled after all other low priority processes have consumed their time-slice. At that moment it starts exchanging on-chip registers for off-chip registers. Meanwhile, it adjusts the hardware scheduling list, so that it continues to hold on-chip processes only. Having accomplished this, the process manager puts itself at the end of the low priority scheduling list, followed by a forced context switch. The newly allocated on-chip processes can now run until the next scheduling round arrives.

Process Administration

The transputer hardware scheduling list only keeps track of on-chip runable processes. We have used the following structures to manage the rest.

free lists. Two lists are used to track stopped and empty processes. One for the on-chip processes and one for the off-chip processes. Both lists are needed to allow reuse of unused process space. This decreases the number of garbage collections and it is important for fast process allocation. When a process is allocated, it is taken from one of these lists, and if this is not possible it is allocated from the heap, which is more costly, mainly caused by additional initialisations. When a process stops - and de-allocates - itself, it will have been running on-chip, so it puts itself in the on-chip list. The process manager moves stopped processes from the on-chip list to the off-chip list. The latter does not contain empty processes, as these are not evacuated.

suspended list: A doubly linked list keeps track of all off-chip suspended processes. No such list exists for on-chip suspended processes. If a process suspends itself, it will have been running on-chip and it simply places a ‘suspended’ mark in its process id (the *pid* in figure 3-2). The process manager scans the on-chip memory for suspended ones and puts them in the off-chip list. On waking up an off-chip suspended process it will be placed in the off-chip runable list (see below). If it is still on-chip it can be started directly after clearing the suspended mark, by inserting it in the transputer hardware scheduling list.

runable list: this is the off-chip variant of the transputer scheduling list. The process manager moves runable processes between this list and on-chip memory.

We try to swap registers in a sensible way, so that processor time does not get wasted. The policy for replacing processes is explained below. To understand this we first need to clarify the states a process can have:

- *runable:* Runable processes are live processes that are able to proceed computing something. Whenever a processor has runable processes, it will be running one of them, which is then the running process.
- *suspended:* Suspended processes - also known as sleeping ones - are live processes waiting for some reviving result. They cannot proceed without it. As they are not able to do anything they should not take up processing time.
- *stopped:* Stopped processes are carcasses. They have registers, stacks and a heap, but they are not running nor suspended. They are dead.
- *empty:* Empty processes are not even carcasses, they are skeletons consisting of merely an on-chip register set (without stacks and without a heap). This happens to an on-chip stopped process when the garbage collector hits it. It is not exactly a process, but rather a left-over part of it.

On-chip processes are replaced by off-chip *runable* processes only. They are removed in the following order:

1. Stopped and empty processes are removed first. The empty ones are simplest. They are not really moved to off-chip memory but plainly overwritten by the registers of off-chip processes. The stopped ones on the other hand, are evacuated, but only the stack pointers and the heap pointers are saved.
2. Suspended processes are removed next, provided there still are off-chip runable processes. All registers are saved.
3. Runable processes are evacuated last, provided there still are off-chip runable processes. All registers are saved.

Depending on the number of runable processes the overall overheads of context switching will vary. If all runable processes fit in internal memory there are no costs apart from the hardware context switching time, which is negligible on a transputer (less than one microsecond). As the number of runable processes increases the overheads grow until there are twice as many processes as fit in internal memory. Every scheduling round will then replace all on-chip processes. These costs are comparable to the costs of saving and restoring registers on an architecture with a single register set.

Still, the duration of a time-slice is much higher than context switching time: some milliseconds compared to a few microseconds at most. Processes rarely run for shorter periods of time, partly because we avoid polling, in contrast to van Groningen (1992). Usually, the costs of sequential processing remain dominant.

3.5.2. Hardware context switches and registers

The main reason to withhold general purpose registers from the transputer design has been that saving registers during context switches would increase context switching times. While this may be true for common sequential hardware, we think it is very well possible to develop an architecture that combines general purpose registers with fast context switching. This can easily be explained by looking at the process manager presented earlier.

The process manager ensures that part of a process is cached in on-chip memory during execution. This software solution runs interleaved with normal processing, so it takes time. However, a hardware process manager could run in parallel with other processes. If special register caches are supported, saving the registers of the previously running process can delayed and performed during execution of the current one. Likewise, the registers of the next process can be loaded in advance. In this way the hardware anticipates context switches. It is not unthinkable such mechanisms will be incorporated in future general purpose hardware, as pre-emptive scheduling has found its way to desktop systems, mainly driven by the extensive use of graphical user interfaces.

One can extend this concept to other parts of a process, such as its stacks and the private heap. Part of these can be moved between the caches and main memory in a similar way when a context switch occurs. This is a general notion. One does not need to tailor hardware for functional languages. It is sufficient to provide a way to program memory management hardware so that it can move data in parallel with execution of other processes. In addition to commonly used separate functional units for integer arithmetic, floating point arithmetic and branching, one would need a functional unit for moving data. In a limited way, this is already being provided by caching mechanisms that allow background manipulation (flushing) of cache lines.

Concluding, we find that the availability of registers is crucial for good parallel performance. The disadvantages are few. Context switching times will only become a bottleneck when processes run for extremely short periods of time. This rarely occurs. Normally sequential processing speed is more dominant. In addition, appropriate hardware can considerably reduce the costs of saving and restoring registers.

3.6. Nodes

Efficient graph reduction requires a good node representation. As a rule of thumb the most compact representation is best. Firstly, the memory requirements of a functional program can be more of a bottleneck than speed of execution. Nodes that are unnecessary large may prevent execution of programs due to shortage of memory. Furthermore, smaller nodes imply cheaper graph rewriting, for less fields need to be initialised during node construction and less garbage collections are needed. All the same, one should be careful not to simplify nodes too much. It is necessary to keep possible future developments in mind. At least during development of the language it must remain possible to incorporate new techniques without substantial changes to the node layout.

Waiting lists and locks form the most important difference between the nodes used in a parallel implementation and those used in a sequential one. A locking mechanism suspends processes if they try to evaluate a node that is already being processed. Besides avoiding unnecessary work, locks regulate the interaction between producing and consuming processes. Waiting lists are related to locking. In contrast to polling locked nodes, they administer suspended processes in the locked node, so that they do not consume any processing time. This is important if many processes are running on a large network with expensive communications. On average, many of them will be suspended and waiting lists can be used to keep the overheads to a minimum. A small drawback is the

need to store waiting lists in the nodes and to unlock them explicitly. The latter will be dealt with in the next chapter.

3.6.1. Node usage

Node design requires careful consideration of the way nodes are used during reduction. The abstract parallel ABC machine does not take this into account and uses a rather general and spacious representation for its nodes. They are split in a fixed and a variable sized part, allowing to update nodes in place, instead of using indirection nodes. This simplifies node access, in particular during pattern matching. The variable sized part contains the arguments of a node, whereas the fixed sized part contains the following four fields:

- *descriptor*. This is a representation of a Concurrent Clean symbol. It contains information about the arity of the node and a string representation of the symbol. Additionally, it provides information for applying partial application nodes (see also chapter 6) to additional arguments.
- *code*. This field refers to the code that evaluates the node to Root Normal Form (RNF). During reduction this code pointer will change. Nodes that are being reduced refer to code that will suspend any reducer. Thus nodes can be locked. Evaluated nodes on the other hand refer to a return instruction.
- *waiting list.* This is a list of suspended processes that are waiting for the result of the locked node.
- *argument pointer:* a pointer to the variable sized part, containing the arguments.

If we focus on they way the parallel ABC machine uses its nodes, we can distinguish two classes: constructors and function nodes. Constructors simply consist of the nodes in RNF and function nodes take up the rest. Empty nodes are special function nodes. Examination of the way that node fields are used during normal reduction reveals the following properties.

1. With respect to the code and descriptor fields:
   a) All constructors contain code that immediately returns, because the node has already been evaluated.
   b) The descriptors of function nodes are accessed infrequently. Only algorithms that interpret nodes, such as the garbage collector, need this information. The descriptors of constructors on the other hand, are accessed frequently, in particular during pattern matching.
2. With respect to locking:
   a) Function nodes can be locked by placing a special evaluation code in it that blocks processes. Constructors are never locked.
   b) Processes get suspended on locked nodes only. If a process starts to reduce a locked node, the evaluation code automatically suspends the process and inserts it in the waiting list.
   c) Before overwriting a locked node by an unlocked one, all processes waiting on the locked node are released and its waiting list is emptied.

d) A locked function node does not need any arguments. Empty nodes on the one hand, are locked function nodes of zero arity, and function nodes on the other hand, can only become locked after pushing their arguments on the a-stack. Hereafter these arguments will never be accessed via the node itself, but only through the a-stack.

3. With respect to updating: only function nodes risk to be overwritten, and except for empty nodes, they are always overwritten by constructors. An empty node with a non-empty waiting list cannot be overwritten by a function node either. Instead, it will be updated by the result of the evaluated function node, which is a constructor.

3.6.2. Basic node structure

These properties have enabled us to deviate substantially from the node representation used by the parallel ABC machine. First of all, we have realised a combination of the code field and the descriptor field, based on the first two properties. We have removed the code field from constructor nodes, and the descriptor field from function nodes. The descriptor of a function can still be accessed indirectly via its code (see figure 3-6). A tag has been added to each node to discriminate between functions and constructors. This will be discussed in more detail later.

The following four properties make it possible to maintain waiting lists without sacrificing memory. 2a, 2b and 2c imply no waiting list field is needed for constructors nor for unlocked function nodes. 2d states that locked function nodes do not need any arguments. Consequently, one of the (former) argument fields can be used to store the waiting list when a function node becomes locked. The rest of the arguments will be dispersed with, so that memory leaks caused by irrelevant arguments are avoided.

The remaining property effected a further reduction in actual node size. Constructors do not need a minimum size for their fixed part because they will never be overwritten. Function nodes on the other hand do need a minimum size - the maximum size of the fixed part of a constructor -, but they do not have to be split. Only empty nodes risk to be overwritten by function nodes and in this case the compiler is always able to ensure that they are big enough.

The picture below elucidates the resulting node structures. The distinct layout of function nodes of arity zero allows the creation of arbitrary large empty nodes. Constructors need a variable sized part only if they contain more than two arguments. We have chosen this particular limit because some important constructors have less than three arguments (list constructors and binary tree constructors, for instance), while function nodes commonly have more than one argument. It should not be higher, to avoid increasing the minimum size of function nodes and spilling too much memory.

To enable the construction of special function nodes the garbage collector preserves the contents of empty fields. Consequently, function nodes of arity one may hold one word of extra information, while function nodes of arity zero may incorporate an arbitrary number of additional data fields. Amongst others, we exploited this feature for the construction of channel nodes and waiting list elements. These will be treated in more detail later.

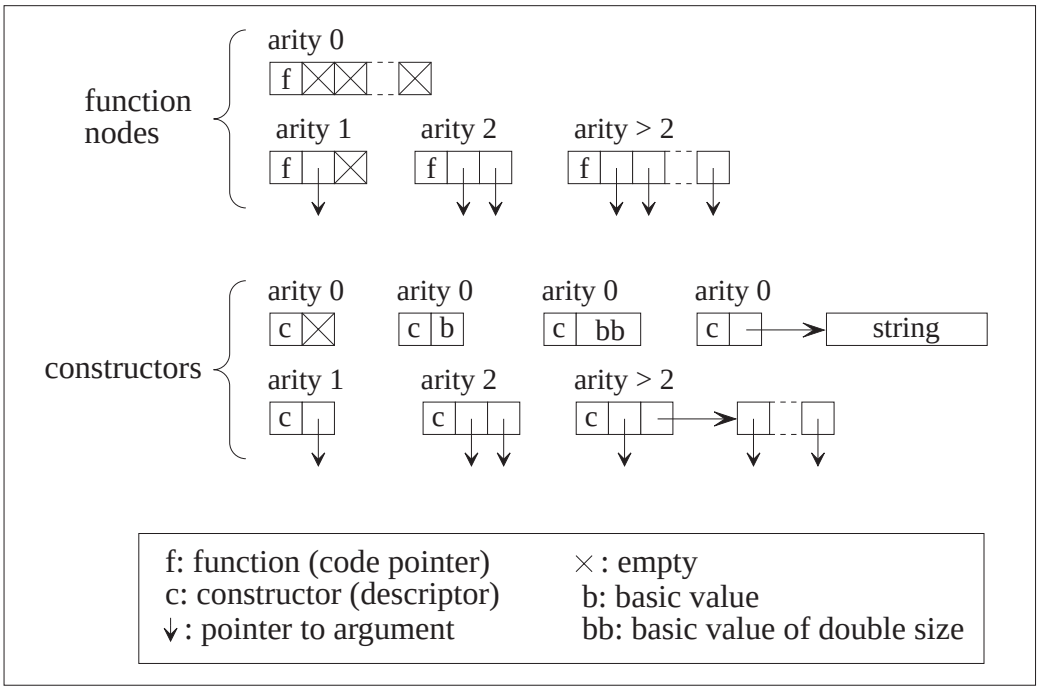


Figure 3-3: *Node layout, tailored to the particular use of each node.*

As we can see in figure 3-3, the exact structure of a node is completely determined by its kind (constructor or function) and its arity, except for nodes of arity zero. Consequently, processes that interpret nodes, generally need merely two tests for extracting the structure of a node. They need to make exceptions for nodes of arity zero only. To decrease the overheads of testing for alternatives still further, constructors that only contain a descriptor field occupy two words instead of just one.

As a result of all this, an interpreting garbage collector will not necessarily perform worse than one that uses special garbage collecting code fragments that are associated with each node. The overheads for extracting and calling such code should not be underestimated, compared to testing for the kind and size of a node. With the current node layout the overheads are roughly the same. Note also that an interpreting garbage collector is likely to fit in the processor cache (if it would exist in transputer hardware), while it also allows for easy experimentation. Therefore we currently prefer interpretation of nodes to implement garbage collection.

3.6.3. Waiting lists

Comparing this node layout to the one employed in sequential implementations of Clean we see there is no difference. No special nodes are needed to accommodate waiting lists. If a function node gets locked, all arguments are removed and it simply transforms into an empty node, that is, a locked function node of arity zero. If then later a process tries to evaluate the empty node it turns into a locked function node of arity one, with the waiting list as its argument. Note that locked nodes with a waiting list do not need to be bigger than the minimum size, because they cannot be overwritten by a large function node, but only

by its result. This also means that a large empty node (arity zero) can be transformed into a small one (arity one) if a process suspends on it.

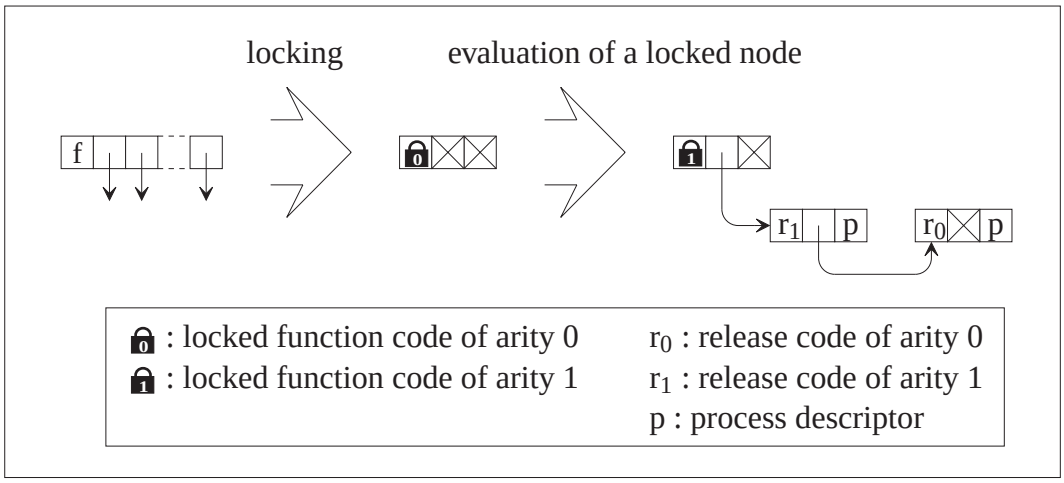


Figure 3-4: *Locking and waiting lists.*

The elements of a waiting list are special function nodes of arity one, except for the last one, which has arity zero. Each contains information about a single suspended process. This is stored in the extra data field. Processes will be woken up by evaluating each waiting list element.

This allows some freedom to experiment with different forms of evaluation. Various actions can be performed on updating a node. For instance, instead of suspending, processes may also proceed with other computations after accessing a locked node. They can leave a special waiting list element behind that will notify the process when the node becomes updated. Another use would be to track updating of certain nodes during debugging.

3.6.4. Tags

Combining the code field and the descriptor field makes it necessary to distinguish constructors from function nodes during reduction. Some kind of tag must be added, without increasing node size. The most suitable way depends on the hardware. The transputer has a signed address space, so pointers are always negative if the total amount of memory does not exceed 2 Gigabyte. By implementing descriptors as positive offsets into a descriptor table we could distinguish them from negative code pointers.

As a result, the most significant bit of the code/descriptor field indicates whether a node has been reduced to Root Normal Form or not. Whenever we need an evaluated node, we first check this bit and if is one, we call the evaluation code stored in the node. Otherwise we know the node has already been evaluated and we jump around the call (figure 3-5).

If we merely consider the basic costs of a calling a subroutine compared to the basic costs of testing and calling we might argue that it is better to avoid the use of tags. However, in functional programs, most of the time that processes try to evaluate nodes they are in RNF already. This is known as the 70 percent rule of Augustsson and Johnsson

(1989). As a result, the costs of both solutions are about the same on average (Kingdon, Lester and Burn, 1991). In addition, these are the minimal costs. In reality the costs for performing subroutine calls and jumps are higher as these cause pipe-line breaks. This makes testing tags relatively cheap as it reduces the number of pipe-line breaks. Note also that a growing number of modern processors - unlike the transputer - incorporates sophisticated instruction pipelining features that may hide the costs of (conditional) jumps. Doing the same for calling code that is stored in a node is more difficult. And finally, as we can see above, after testing the tag of an evaluated node, we will jump around quite some code that is needed to set up a call. Depending on the actual use of registers and stacks this includes stack checking code and code to save and restore registers. All this, plus the advantage of using distinct representations for constructors and function nodes, makes tags invaluable for an efficient implementation.

```
    ldl     node_ptr        get pointer to node
    ldnl    code            get code/descriptor field
    mint
    and                     extract most significant bit
    cj      in_RNF          jump on zero (descriptor)

    check_stacks            macro, check if enough space on stack
    save_registers          macro, save registers on stack

    ldl     node_ptr
    ldnl    code
    gcall                   call evaluation code

    restore_registers       macro, restore registers from stack

in_RNF:
```

Figure 3-5: *Evaluating a node.*

3.6.5. Descriptors

The descriptor field of a constructor holds an index in a descriptor table. Each processor contains an copy of this table, which stores a descriptor for every symbol used in the program. A separate entry exists for every arity a symbol can take. The descriptor provides a string representation of the symbol, information about its arity, and code pointers that are used for quickly applying a curried function to an additional argument. In contrast to a sequential implementation, in a parallel one each descriptor additionally contains a pointer to the evaluation code of function nodes. This is needed to implement graph copying (see chapter 5), but it might also be used to provide some form of dynamic code loading

Storing the descriptor of a constructor directly in the node itself as an index makes it very easy to identify a constructor symbol. This is important for pattern matching. One only needs to compare it to another index, which is a compile-time constant most of the time. Many processors, including the transputer have efficient instructions to do this.

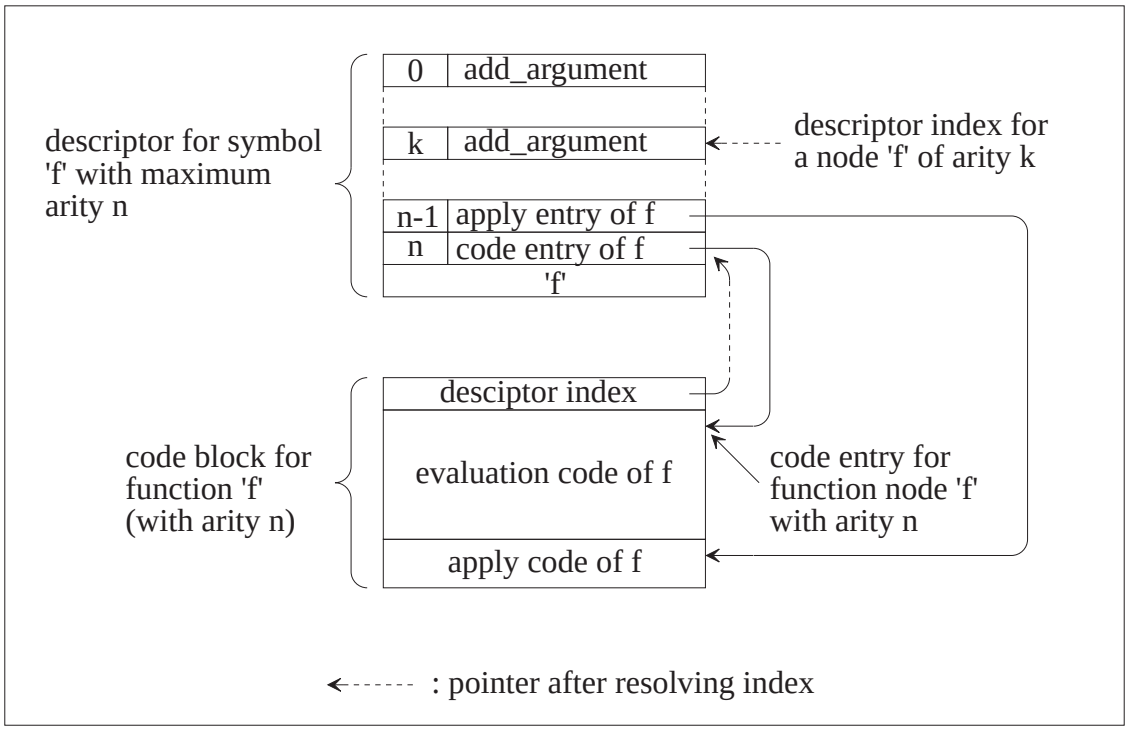


Figure 3-6: *Layout of descriptor and code block.*

Figure 3-6 shows the layout of a descriptor for an arbitrary function. Additionally, it reveals the structure of the corresponding code block. Note that all access to the descriptor table has been based on the use of indices, and that it is possible to get to the descriptor of a function via its code and vice versa. This is used for graph copying and garbage collection, which will be explained in chapter 5.

3.6.6. Smaller nodes

The FAST compiler (Langendoen, 1993) employs a node structure that is even more compact. This has been achieved by using an intricate scheme for tagging pointers and data. As a result, a list of strict integers takes only two words per element. Clearly, this saves a lot of heap space. It has some disadvantages though. First of all, indirection nodes are needed to update nodes and secondly, basic values are tagged and have a uniform size. Even if the computational overheads of the latter are not significant, as the authors claim, it is questionable whether the remaining space (31 bits for floating point numbers) provides sufficient accuracy during computations. More worrying however is the lack of type information in data structures. Not only does this make debugging and monitoring harder, especially during language development, it is also unclear how such a node representation can be used in conjunction with more sophisticated language features like existential types and overloading. These may require runtime checking of types, or some tag that identifies the operations that are possible on a particular type of node. To avoid hampering such developments we have chosen to stay with the current node layout.

4. Code Generation for the Transputer Processor

In this chapter we will explain how one can compile (parallel) ABC code to efficient code for a *single* transputer processor. For now, we restrict ourselves to the interleaved execution of multiple processes on one transputer. In the next chapter we will see how to support true parallelism on a multi-transputer system.

First we will focus on the generation of code that does not support interleaved execution in any way. That is, we will start with a *sequential* implementation for a single transputer processor. We will not treat generation of sequential code in detail, but we will show the relation between code generation for a single transputer processor and for a conventional register-based sequential processor (such as a Sparc, or a Motorola 680x0). It will become clear that the differences can be kept to a minimum, even for a rather divergent architecture like the transputer processor, which does not have any true general purpose registers at all (see section 1.6).

Having seen in which way the transputer hardware influences code generation compared to more common architectures, the following part of this chapter will examine the consequences of supporting *interleaved* execution of multiple processes on a single transputer processor: context switches, locks and waiting lists, and less evident, the handling of stack overflows. Performance figures show that these additions do introduce overheads, but that they are tolerable.

And finally, we will relate the sequential performance of our implementation to that of other implementations. We will see that simulating registers does not result in bad performance. In contrast, our performance measurements show that our Concurrent Clean implementation compares favourably to transputer implementations of other functional languages.

This chapter has been organised as follows. In section 4.1 we will determine which target language is most suited to be generated on a transputer system. Section 4.2 will deal with the generation of purely sequential code. The following three sections will present the necessary additions to support interleaved execution, and indicate their effect on performance: section 4.3 will show how to cope with stack overflows; section 4.4 explains the realisation of locks and waiting lists; and in section 4.5, we will see how to support efficient context switching on the transputer. Hereafter, in section 4.6, we will compare the sequential performance figures of our implementation to those of other implementations. The final section will list our main conclusions.

4.1. The target language of the code generator

As we have argued in the first chapter, one cannot stop at generating abstract machine code. Eventually, efficient machine code is required. Generating (imperative) machine code from abstract machine code is easier than doing it directly from a functional program, but it is far from trivial if we want to get optimised code. There are several ways to achieve this, and a basic question in this respect is, which language should be generated. We will examine this below.

4.1.1. The merits of generating C

Portability and ease of code generation are major justifications for generating (generic) C code instead of machine code. We contend this view. Instead, in this section we argue that C provides too low level a language to allow true portability or comfortable code generation.

With respect to portability issues, it is important to note that there is a certain lack of standardisation on the efficiency of C programs. Standard C programs may look portable, but in practice there are many factors that influence performance. The efficiency of a single C program may vary considerably on different architectures. This limits true portability. We will elaborate this below.

C does not abstract from the underlying architecture. It is too explicit. Essential language features cause this problem. Having pointers, it becomes possible to manipulate data in a very explicit way. Consequently, there are optimisations a C compiler simply cannot do. The differences become clear when comparing the efficiency of some programs written in FORTRAN to their counterparts written in C. In some cases the compiler has insufficient information about the locations that pointers refer to. This destroys the possibility for a good analysis on dependencies (for instance, to make reordering of instructions possible). To make up for this deficiency, C allows the use of directives for tuning performance. These do not always have the same effect on all architectures. For example, the standard keyword ‘register’ has no meaning on a transputer. Some directives might even compromise correctness, if the programmer does not use them with caution. In short, C programs will not always result in the best possible code on all architectures.

Moreover, the quality of the code that different C compilers generate varies. This is sometimes caused by the use of custom directives that are not supported by all compilers. At least as important however, is the diversity in the implementation of optimisations. There is no ground for assuming that a particular C compiler provides all optimisations needed to generate the best possible code. It seems that some C compilers have come to trust programmers to write the most efficient code for a particular machine, based on the explicit manipulation of performance that is inherent in the language.

Compilers that translate functional programs to C depend heavily on the (absence of) features of specific compilers and machines. They typically rely on the ability of the compiler and the machine to keep certain global variables in registers. In addition, they commonly generate a special kind of C program that incorporates explicit manipulations of stack- and heap-elements. This kind of code is very specific about how computations are to be done. It does not contain much high-level information and it rather looks like assembly language with C syntax than like a program written in C.

Some compilers have serious problems compiling such unusual code, let alone optimising it for a particular machine. For example, if we consider the code that C compilers would produce for the transputer, we see it contains severe inefficiencies. On the one hand, important global variables end up in the global data area and not in a register, because the transputer does not have any (see also section 1.6 and section 3.3). This makes accessing these values very expensive. On the other hand, if we generate C that explicitly manipulates the heap and the stack in the same way that implementations for register-based machines do, poor transputer code will be produced. Suppose we generate C code that adjusts the stack pointer incrementally within a single basic block, which is common practice in C programs for register based machines. Some processors, like the Motorola 68000, support this kind of addressing well and C compilers will generate excellent code for these machines. Unfortunately, the transputer does not support this way of addressing, and its C compilers do not combine several small stack adjustments into a large one, which would have been best. Thus, one cannot expect the same C program to run equally well on all sorts of machines. With respect to efficiency, C is not sufficiently portable.

The main advantage of generating C that remains appears to be ease of code generation. However, the merits of this are not as clear as one might expect. We feel that the - relatively small - additional efforts needed to generate machine code instead of assembly-like C are worth the reduction in compilation time in general. In particular for the transputer, generating C will not be much easier than generating assembly, because in both cases registers need to be simulated explicitly. One does not get many optimisations for free. Furthermore, looking at this from another direction, we have experienced that adapting a sequential register-based code generator to the transputer was not extremely difficult (as we will see in section 4.2). Transitions from one sequential register-based machine to the other proved to be relatively easy as well, that is, as far as code generation is concerned. It turned out that most work is not related to adapting the code generator, but to adjusting runtime systems (including I/O systems) to the peculiarities of the underlying operating system. Clearly, more code generation problems will arise if architectures emerge that are radically different, but this also holds when C is generated.

4.1.2. The merits of generating Occam

Another candidate object language is Occam, which has been promoted by INMOS as a high level transputer assembly language. Indeed, all Occam constructs can almost directly be mapped on the transputer hardware, so *if* a solution can be expressed concisely in Occam, it can be compiled to extremely efficient transputer code. However, the converse is not the case: not all assembly code can be expressed concisely in Occam. Occam lacks the freedom that is common in assembly or languages like C. Recursion, dynamic memory allocation, dynamic process allocation and non-flat data structures (using pointers) have never been a part of Occam. This is another reason that Occam is so efficient: due to language restrictions the Occam compiler is able to perform many compile-time checks and optimisations. It not only avoids runtime overheads, but it also provides a safe parallel programming environment. For instance, it is able to statically check array bounds and detect sharing of variables.

Lack of features found in languages like C may not be a problem for the main market that Occam has aimed at, namely that of embedded control systems, but for implementing a system like Clean it is less suited. Several essential features need to be simulated and this has consequences for performance. The figures presented for SkelML, which is a parallel implementation of standard (strict) ML that compiles to OCCAM, show this (Bratvold, 1993, 1994). SkelML is more than a factor 3 slower than the Clean implementation (see table 4-8 and table 5-4).

4.1.3. generating transputer assembly

As we have argued above, generating C (or another 'high level' imperative language) is not always a workable alternative. Especially for the transputer, generating C or Occam will not solve many problems with respect to code optimisation. If one does not generate assembly-like Occam or C, these languages introduce considerable inefficiencies. Consequently, we will directly compile parallel ABC code to efficient transputer assembly code. The remaining part of this chapter will focus on problems that are related to achieving this goal.

4.2. Sequential code generation

Sequential compiler technology forms the backbone of our implementation. However, constructing an efficient sequential implementation is not our main concern here. This already has been focused on in (Plasmeijer and van Eekelen, 1993; Nöcker *et al*, 1991-a, 1993-a, and 1993-b; Smetsers *et al*., 1989, 1991, 1993; van Groningen *et al*., 1991). In this section, we will take a sequential implementation as a starting point and transform it to a sequential transputer implementation. The following sections will show how this can be extended to an efficient implementation that supports interleaved execution of processes.

To a large extent, sequential compiler technology is based on the efficient use of registers and stacks. In the previous chapter we have chosen our data structures in such a way that it remains possible to use this way of code generation for the transputer. As a result, sequential code generation for the transputer does not differ substantially from sequential code generation for a register-based processor. The most profound change is caused by the real transputer registers. Having this small hardware evaluation stack it becomes possible to pass intermediate results on it - at times no context switches are possible -, instead of storing them in pseudo-registers or on a conventional stack. As a result, transputer programs will use fewer (pseudo) registers on average than register based machines. This section will explain the consequences for sequential code generation. We will start with a short overview before proceeding with the details.

Ordinary sequential code generation for register-based machines is divided into a number of phases (see figure 4-1). The conversion phase partitions the ABC program into basic blocks and converts each block to a directed acyclic graph (DAG) representation. Each DAG contains the data dependencies between the stack entries at the start and at the end of its basic block. The global register assignment phase takes each block (in DAG form) and determines which stack entries should be placed in registers at the beginning and at the end. The ordering phase then derives the order in which the operations in the DAG

should be evaluated. This is followed by the code generation phase that generates intermediate code in the given order. The main difference with the final code is that there is no bound on the number of registers. This means that the intermediate code might use more registers than are actually available. This is solved in the last phase. We will now take a closer look at these phases.

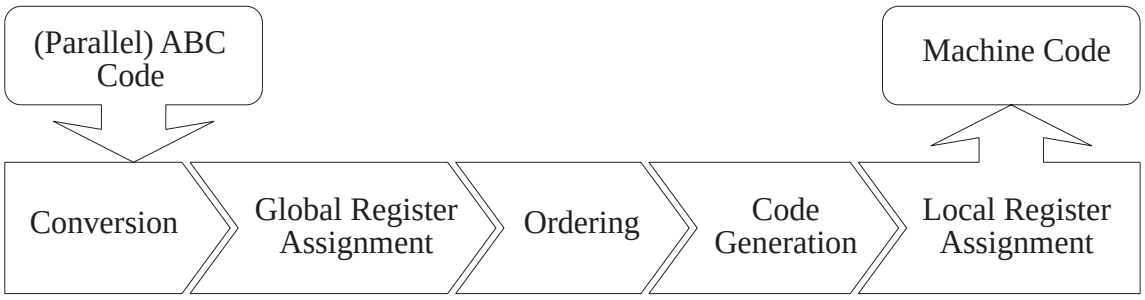


Figure 4-1: *The code generation phases*

4.2.1. The conversion phase

The conversion phase constructs a DAG that represents the operations needed to get from the start of a basic block to its end. The leafs either contain constants or load operations on stack entries at the beginning of the basic block (the arguments of the basic block). The root nodes store the results of the basic block on the stack. Intermediate nodes represent the operations that are needed to compute these results from the arguments. The DAG is constructed in such a way that unnecessary dependencies between instructions are avoided. Basically, all dependencies in it are data dependencies.

4.2.2. The global register assignment phase

The global register assignment phase determines which stack entries should be kept in registers at the beginning and the end of each block. It replaces certain *store stack* and *load stack* nodes by *store register* and *load register* nodes respectively. In addition, it combines information of adjacent blocks to optimise stack access (for example, it removes garbage stack entries as soon as possible). This may change the DAGs for a combined group of basic blocks, so the next phases are postponed until the global register allocation phase for the group has ended.

These first two phases are essentially identical for both the parallel and the sequential implementations of Concurrent Clean that have been developed at the university of Nijmegen. This is in particular convenient, because they comprise elaborate machine independent optimisations. The next phase is the first one that diverges. It determines the order in which the operations in the DAGs should be performed. For the transputer this is slightly different, because the hardware evaluation stack influences this order.

4.2.3. The ordering phase

The code generator may generate instructions in a different order than the order of the PABC instructions, as long as the final result of each basic block remains the same. Part of

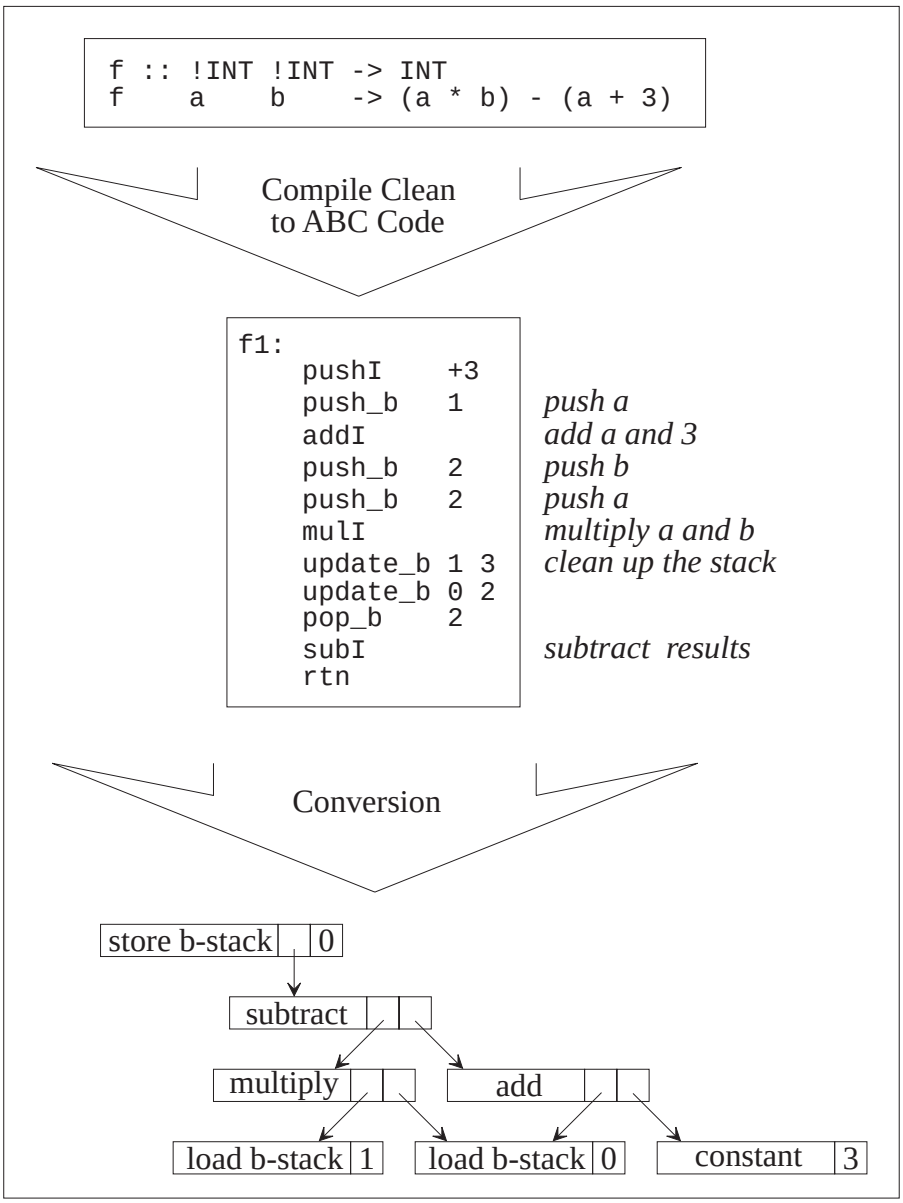


Figure 4-2: *Constructing a DAG from a function*

this order is fixed: the arguments of a node need to be evaluated before the node itself. The evaluation order of the arguments remains to be determined. This is done by the ordering phase.

For register-based machines the ordering phase tries to minimise register use, as this is an important but scarce resource. This is a rather complex problem. Sub-expressions may be shared, and this makes finding an optimal solution expensive, because one cannot compute the register needs of a single expression independently of the evaluation order of the other expressions. All possible permutations need to be tried, which is *O(n!)*. This is too expensive in practice, so instead the sequential Concurrent Clean implementations a safe estimation of the costs (Plasmeijer and van Eekelen, 1993).

The transputer additionally has a small evaluation stack. The best evaluation order for such a stack is to evaluate the expressions that use the most stack space first. This avoids stack overflows and increases the probability that intermediate results can be passed on the evaluation stack instead of in pseudo-registers.

Combining the ordering criteria for the registers and the evaluation stack, we let register use prevail over evaluation stack use. The reason for this is that in general, the evaluation stack use of a particular sub-expression does not say much about its use of registers, as register use largely depends on sharing of sub-expressions. There is no clear relation between evaluation stack use and sharing. This means that a good ordering according to evaluation stack use could easily result in a very bad ordering according to register use. Register use on the other hand, does give some useful information on evaluation stack use: inefficient evaluation stack use may increase the register use. Although possible, it is not very likely that the evaluation stack is used inefficiently when register use is good, and even if this occurs, avoiding a worse register allocation is more valuable in general (see also the frame ‘Ordering Dyadic Operators’).

The evaluation stack use will only play a role when the register uses of different orderings are equal. If this is the case the alternative with the largest evaluation stack use will be evaluated first. If the evaluation stack uses are the same we will evaluate the arguments of a node from left to right, as non-commutative operators are constructed in such a way that it is slightly more advantageous to evaluate arguments in this order.

In brief, the ordering phase for the transputer will basically determine the evaluation order in the same way as for register-based machines. Differences only arise in case a pure register implementation cannot decide which order to choose.

4.2.4. The code generation phase

The code generation phase produces intermediate code for the nodes in the DAG in the order that has been derived by the previous phase. This code resembles the final code, but it assumes an unlimited number of registers is available. The latter will be solved in the last phase.

Different code needs to be generated for different machines. One might expect that profound changes are needed while porting the code generator to another register-based architecture, but often this is not the case. The nodes in the DAGs represent basic operations. These operations are commonly found in modern general purpose processors. Furthermore, addressing takes a similar form for these machines. The arguments of each instruction typically consist of registers and constants. Additionally, offsets can be used. Sometimes extras have been added such as post-incrementing the contents of registers. This makes code generation for many register-based machines fairly similar.

Even for the transputer, code generation proceeds in the same way. The processor provides the same basic set of operations that register-based machines supply. Addressing seems somewhat different however. The arguments are not in registers, but on the evaluation stack. Still, loading the evaluation stack can only be achieved via the workspace pointer or by loading a constant. The workspace pointer in turn, refers to the emulated register set of a process. So eventually we are using registers and constants as well.

Ordering Dyadic Operators

The ordering algorithm for dyadic operators is listed below. It first calculates the evaluation stack use and register use for both arguments in a recursive way (*use1* and *use2*). This includes the number of registers needed for evaluating each sub-expression (*arg.reg_use*) and the increase in register use this causes (*arg.increase*). The latter indicates the change in register allocation that the evaluation of the sub-expression effects. It will be small - possibly negative - when the expression uses many shared values for the last time. Conversely, it will be large when the expression produces many shared values.

Next, the algorithm computes the overall increase in register use (*use.increase*), which is basically the sum of the increments of the arguments. Combined with the argument register uses this value determines the register costs for both possible orderings (*order1_use* and *order2_use*). The order that needs the least registers will be chosen (*arg1_first*). If the costs are the same the use of the hardware evaluation stack (*arg.stack_use*) determines the evaluation order.

Note that if no sharing exists, the increase in register use will be zero, so that the overall register use is largely determined by the maximum of the register uses of the arguments. This is the same for both orderings. As a result the evaluation stack use dominates in a system without sharing. On the other hand, if sharing exists avoiding inefficient use of registers will be more important than using the evaluation stack in an optimal way, for storing intermediate results in registers is inexpensive, compared to dealing with register overflows, so that values need to be stored on a regular stack.

```
calculate_dyadic_register_use :: Graph -> RegisterUse
calculate_dyadic_register_use operator = use
where
  arg1 = first_argument_of operator
  arg2 = second_argument_of operator
  use1 = calculate_register_use arg1
  use2 = calculate_register_use arg2

  use.increase = use1.increase+use2.increase+(toInt(is_shared operator))

  order1_use = maximum order1_use' use.increase
  where order1_use'
    | use2.stack_use >= stack_size && (not (is_shared arg1))
       = maximum use1.reg_use (use1.increase + 1 + use2.reg_use)
    | otherwise
       = maximum use1.reg_use (use1.increase + use2.reg_use)

  /* order2_use is defined similarly */

  arg1_first
    | order1_use == order2_use = use1.stack_use >= use2.stack_use
    | otherwise                = order1_use < order2_use

  use.stack_use = minimum stack_use' stack_size
  where stack_use'
    | arg1_first = maximum use1.stack_use (1 + use2.stack_use)
    | otherwise  = maximum use2.stack_use (1 + use1.stack_use)

  use.reg_use
    | arg1_first = order1_use
    | otherwise  = order2_use
```

Thus, we can view small groups of transputer instructions as simulating the behaviour of ordinary register-based instructions. Each group is delimited by the points where the evaluation stack is empty. It starts by loading the evaluation stack with the arguments of the instruction it mimics. These are either constants or the contents of registers. Extra transputer instructions add offsets if necessary. Having set up the input parameters the basic operation can be performed, followed by storing the result.

Simulating every register-based instruction separately does not give the best performance. As stated earlier the transputer allows us to pass results on the evaluation stack, instead of in registers. To achieve this the code generator will never store the result of a basic operation in a register, unless a stack overflow is imminent, a context switch is about to occur, or when sharing dictates it. This increases the group size. In a sense we are constructing new register-based instructions of arbitrary size.

4.2.5. The local register assignment phase

The intermediate code is allowed to use more registers than are actually available. The last phase solves this. We have tried to minimise register use in the previous phases, so it is not needed very often. If necessary, it maps the registers used in the intermediate code to the real emulated registers of our implementation.

A mechanism similar to paging is used for this (Plasmeijer and van Eekelen, 1993). The local register allocator replaces every instruction that uses a virtual register by one that uses a real register. If at some point all physical registers are in use, it generates extra instructions to save one to memory (on the stack, or in the heap). It evacuates the one whose contents will not be used for the longest time (this can be determined by inspection of the generated code). When the contents are needed of a virtual register that has been saved before, the register allocator inserts additional instructions to reload this value in a real register. This mechanism is used on all registers, so that allocation is most flexible. This means that the heap pointers and the stack pointers may be saved temporarily in memory as well.

Doing this for the transputer has both advantages and disadvantages. On the one hand, finding and adjusting instructions that use a register is easy. One does not have to take into account many instructions as there are only three: *ldl*, *stl* and *ldlp*. On the other hand, inserting instructions to save and restore registers is more difficult, because these use space on the evaluation stack and this may cause an overflow. It is here that we actively use instruction grouping. Instructions are only inserted between groups. At these places the evaluation stack is empty, so we can use all stack space.

At the moment we assume that groups do not need more registers than are actually available. This will be valid for most programs. Often, the local register allocation phase itself is not even needed. In case this assumption is not true, a group needs to be split up further, so that register allocating instructions can be inserted within a group.

4.2.6. An example

The next figure shows the code that is generated from the DAG in figure 4-2. It lists both transputer code and MC68020 code. This example illustrates the primary differences - and the similarities - between transputer code and code for common register-based machines.

The local register assignment phase has not been invoked in this case, because few registers are used.

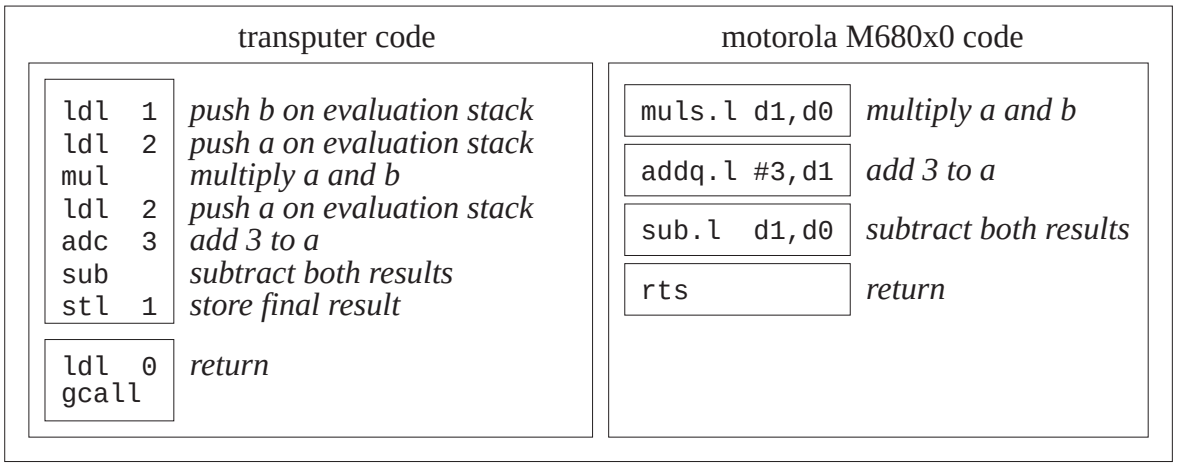


Figure 4-3: *A Small Code Example*

All transputer instructions are merely one byte long, except for the *mul* instruction, which takes two bytes. In comparison, the Motorola code takes two bytes per instruction. Instruction groups have been enclosed in small boxes. Passing intermediate results on the transputer evaluation stack reduces the number of instruction groups to two. Note that after pushing *a* and *b* on the evaluation stack for the last time, the corresponding registers are free to be used for different computations. In contrast, the Motorola code needs *d0* and *d1* to store intermediate results until the subtract instruction at the end.

4.3. Handling stack overflows

Above, we have seen how to generate sequential code, which only supports a single process per processor. Concurrent Clean however, allows many processes to be started on a single processor by means of the ***{I}*** annotation (and the ***{P}*** annotation, if it starts a new process on a processor that already runs one or more processes). These processes should run interleaved and scheduling must be fair. In this section, and the following ones, we will introduce the adjustments that are needed to support this form of processing. In addition, we will show the effects on performance that are caused by these adjustments.

This section will consider the handling of stacks. Having many processes as opposed to one in a sequential language, one cannot simply allocate a huge stack for each process to avoid stack overflows. This would consume too much memory. Even if enough memory were available to give every process a large stack, much precious space would needlessly be wasted on stacks, resulting in superfluous garbage collections. Often, processes do not need a large stack, so it would be better to allot more memory space to the heap instead. As a result, processes should be given a small stack that is checked on overflow and expanded if necessary.

4.3.1. Combining stack checks

Some architectures allow stack overflows to be detected by hardware, for instance by a memory management unit. The transputer does not have such hardware. Instead, the code generator needs to insert stack checking code.

Checking a stack can be costly if it is needed too often. We combine stack checks to some extent, but the abstract ABC code does not provide enough information to limit it to one check per function call in an easy way. It is not entirely trivial to let the Clean to ABC compiler pass this information, as it does not know the exact stack use. Low level code optimisations may affect it. This is a drawback of using abstract ABC code: information does not flow back up. Instead, the code generator needs to re-obtain information about the dependencies between blocks of code. We have not implemented this, so our code is not yet optimal with respect to checking stacks. All the same, if we have a look at the tests below, we see that the current implementation does not incur too much overheads.

Even if sufficient information in code dependencies is available, stacks checks should not always be combined into one per function. First of all, if a function has multiple alternatives, it could check stacks at the start for the maximum use of all alternatives, but if one exists that does not need any stack space this may not be a good strategy. Instead, the stacks should be checked within each alternative. Secondly, one should allow stack size to be decreased occasionally if much of it is unused while free memory is running low. A logical - and efficient - way to do this, is to let the garbage collector decrease the size of a stack when it sees fit (for it can determine overall memory usage and stack usage in a relatively easy way). This may happen at some point after a function has checked its stack, so the garbage collector must somehow ensure that at least the checked amount remains available. A simple way to do this is to limit the amount of stack space that can be checked in one go, so that the garbage collector knows by what amount it can safely decrease the stack size. This means that several checks are necessary if the total stack use of a function exceeds this bound.

The transputer code generator combines stack checks as follows. First it determines the stack needs of each basic block. This comprises the increase in stack size caused by evaluation of the block, the amount of stack space allocated by the local register assignment phase to save registers, and - if a basic block ends with a *jsr_eval* instruction or a similar one - the conditional stack use of this instruction (for saving registers; see figure 3-5). Next, it combines checks for basic blocks that are divided by some form of subroutine call. This is possible because the callee returns the stack in the same state as it finds it. The check of a post-subroutine block may then be joined with the check of an earlier one. The code generator visits basic blocks in reverse order, repeatedly shifting checks to previous blocks. This stops when it arrives at the first combinable block, or if it reaches the maximum value that can be checked at once (to allow the garbage collector to decrease stack size; this has not yet been implemented). And finally, conditional checks are combined with unconditional ones if the latter exist. This strategy has lead to the performance figures presented in table 4-1.

Table 4-1: *The costs of checking stacks. In this thesis, all test of our Concurrent Clean implementation have been performed on a network of T800 transputer processors. Each processor runs at 25 MHz and has 4 Mbyte of memory. Unless stated otherwise, heap size has been set at 3 Mbyte and the initial stack size has been set at 100 bytes. Timings have been performed with the transputer clock and are very accurate, as our transputer network is a standalone system. However, small variations in timings exist, as they are slightly influenced by IO processes. These variations are less than 0.1% of the total execution time.*

	stack checks off	stack checks on	overhead
nfib 30	11.2 sec.	12.2 sec.	8.9 %
tak 24 16 8	10.9 sec.	11.4 sec.	4.6 %
queens	42.6 sec.	43.9 sec.	3.1 %
reverse	64.3 sec.	64.4 sec.	0.2 %
fast fourier	12.8 sec.	12.9 sec.	0.8 %
mandelbrot	134.7 sec.	141.5 sec.	5.0 %
raytrace	314.6 sec.	321.1 sec.	2.1 %
sieve	18.3 sec.	19.4 sec.	6.0 %
quicksort	4.7 sec.	4.9 sec.	4.3 %

We can see that the overheads for checking stacks do not exceed 10%. The nfib benchmark suffers the most. This is not surprising, as it is a small recursive function that computes relatively little during each function invocation. The reverse function on the other hand, does not push entries on the stack most of the time. Instead it uses registers so it does not perform many stack checks.

To illustrate the significance of having stack checks: we have tried to run some interleaved concurrent programs - merely using ***{I}*** annotations - on a single processor without stack checks, but we did not succeed in general. Many interleaved processes had to be accommodated on a single processor, and the initial stack size had to be fairly large for each process, so that often memory problems arose. In contrast, with stack checks, running these programs did not give many problems (but again, a few programs could not be run because of memory shortage). The ‘interleaved’ parallel programs that we *were* able to run without stack checks revealed similar overheads for stack checking as their sequential counterparts.

4.3.2. Expanding a stack

The costs of a single stack check are clear. The two stacks grow from the ends of a single memory block towards each other, so that it takes a single pointer comparison to check both. In addition, the stack pointers can be accessed quickly because they are stored in internal memory. One cannot improve much on this, no matter how stacks are implemented.

The costs of various ways to expand stacks are less obvious. There are basically two methods. One approach uses a stack that consists of several pieces that are linked together. If this segmented stack turns out to be too small, an new part is allocated and linked to the old stack. The other solution uses a traditional monolithic stack that is reallocated when needed.

We use the latter. It involves copying the old stack and at first sight this seems to be expensive. However, it simplifies garbage collection - and debugging - as the stack layout is less complicated. Secondly, most processors allow contiguous blocks of memory to be copied rather quickly. But most importantly, stack reallocations are needed very infrequently. Once a stack has grown sufficiently, many subsequent computations can be performed on it without increasing its size. Note that it is important not to decrease the stack size too quickly, for it may increase the number of succeeding stack reallocations. At the moment we do not decrease it at all, so during the execution of a process we dynamically allocate its maximum stack use (if shrinking stacks are necessary, these can be effectively provided by the garbage collector).

Table 4-2: *The costs of stack reallocations. These tests have been performed with a large initial stack size that does not require any reallocation, and with a small initial stack size. Most programs (except raytrace) are small, but this does not imply that they use little stack space. The raytrace and mandelbrot program need almost 100 Kbyte of stack space, as they convert all pixels to integers and add them recursively on the stack (to avoid overheads for plotting). The fast fourier programs uses about 30 Kbyte of stack space. The others use less than 10 Kbyte. The runtime system increases stacks by about 10% plus a small constant. The latter ensures that very small stacks grow fast enough. Allocating large initial stacks will sometimes increase garbage collection times a little. We have listed the pure execution times without garbage collections as well, to rule out these additional costs. The differences in pure execution times are hardly noticeable.*

	Including Garbage Collection Time		Excluding Garbage Collection Time	
	small stacks (16 bytes)	large stacks (10-100 Kbyte)	small stacks (16 bytes)	large stacks (10-100 Kbyte)
nfib 30	12.2 sec.	12.2 sec.	12.2 sec.	12.2 sec.
tak 24 16 8	11.4 sec.	11.4 sec.	11.4 sec.	11.4 sec.
queens	43.9 sec.	43.9 sec.	42.9 sec.	42.9 sec.
reverse	64.2 sec.	64.4 sec.	59.9 sec.	59.9 sec.
fast fourier	12.9 sec.	12.9 sec.	10.5 sec.	10.4 sec.
mandelbrot	141.5 sec.	141.5 sec.	141.5 sec.	141.5 sec.
raytrace	321.0 sec.	321.1 sec.	320.5 sec.	320.5 sec.
sieve	19.4 sec.	19.4 sec.	19.3 sec.	19.3 sec.
quicksort	4.9 sec.	4.9 sec.	4.7 sec.	4.7 sec.

Table 4-2 shows that this strategy has kept the overheads of stack reallocation to a minimum. The costs can be even less for architectures that allow virtual memory management to be used for realising stack reallocations.

We have not tested a version with segmented stacks. This would require quite some changes to the code generator and the garbage collector. Especially the particularities of the abstract ABC machine make it very difficult to realise an optimal implementation of segmented stacks. Incorporating these would impose too much work for the limited gain they might bring. Segmented stacks do not offer serious advantages because the costs of stack reallocations are negligible. In contrast, segmented stacks may increase overheads if functions frequently cross the boundaries of a segment. This is not unlikely because stack use typically fluctuates quickly. Depending on the complexity of the implementation and the architecture this may be especially disadvantageous in conjunction with caches. Monolithic stacks automatically avoid these problems. They naturally provide the basic features stacks should have, that is, supporting quickly fluctuating memory demands by allowing fast allocation, de-allocation and reuse of memory. On the other hand, linked stacks *can* be more flexible in certain situations, and in principle, they *can* be as efficient as ordinary stacks (provided the right hardware is available). See Appel (1987 and 1994) for a detailed discussion on this subject.

4.4. Handling locks and waiting lists

In contrast to a sequential implementation, a parallel one requires locking of nodes to prevent multiple processes reducing the same node, unless one employs a partially strict reduction strategy that avoids sharing of redexes among processes (Langendoen, 1993). Concurrent Clean uses a lazy reduction strategy by default, and at the moment it always locks nodes[4]. Nonetheless, at places the Concurrent Clean compiler is able to employ strict evaluation, it avoids constructing nodes as much as possible, thereby decreasing locking overheads.

On machines with distributed memory locking itself does not have to introduce any overheads. On entry of a node one simply places a special locking code pointer in it. For comparison, sequential implementations place a special error code in the node to detect cycles in the spine of reduction, and more importantly, to allow the garbage collector to remove unneeded function arguments as soon as possible. The costs are exactly the same.

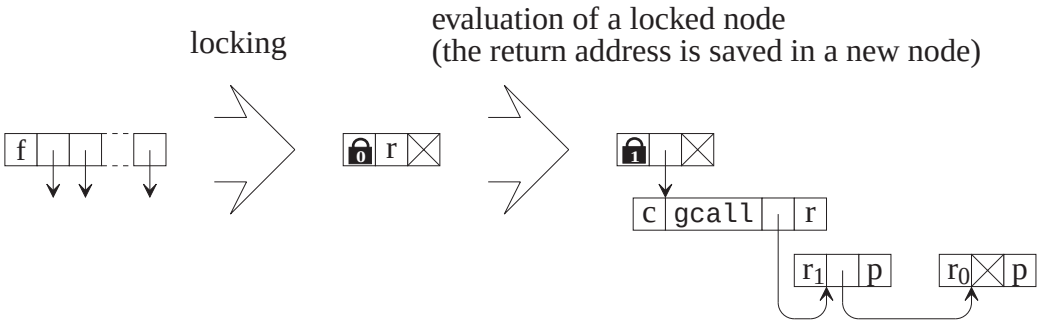
Unlocking a node on the other hand does impose some overhead. On updating a node its waiting list needs to be checked. If it is not empty the processes in it should be woken up by evaluating each waiting list element. So we need an extra test compared to a sequential implementation. Though not seriously affecting performance (see table 4-3) we can improve on this.

In general, updating a node occurs shortly before returning from a *jsr_eval* instruction. Instead of storing the return address on the stack after entering a node, we could equally well store it in the node itself after extracting its arguments. We may use the

4 Regardless of the actual uniqueness properties of nodes. Taking advantage of uniqueness would require different calling conventions. It is not yet clear whether this can be done without hurting performance

waiting list field for this. On the transputer this does not introduce additional costs compared to pushing the return address on the stack. Now before overwriting the node we should retrieve this address and call it after the update has taken place. Ordinary subroutine calls will store the return address on the stack as usual.

This not only saves stack space - possibly avoiding stack checks as well - but it also removes the need to check the waiting list field. Storing the return address in the locked node allows it to be modified when a process tries evaluate the node. Instead of inserting a waiting list, we may first save the original return address and then replace it by code that releases an associated waiting list before returning to the saved address. A possible way to



0 : locked function code of arity 0	r_0 : release code of arity 0
1 : locked function code of arity 1	r_1 : release code of arity 1
c: special constructor (descriptor)	p : process descriptor
`gcall` : transputer gcall instruction	r : return address

All pointers will be made to point to the second field of a node, so that it becomes possible to store code at this position (which additionally enables dynamic loading of code in the heap). The code generator will produce the following code to return from a jrs_eval instruction

```
load_address_from_node
gcall
release_code
```

If the address in the locked node is a plain return address this code will return directly. Otherwise the gcall instruction will jump to the gcall instruction in the node above after storing its return address - i.e. the address of the release code - on the transputer evaluation stack. The second gcall in the node will then return to this code after storing a pointer to the node itself on the evaluation stack. The release code can now retrieve the original return address and the waiting list from this pointer and release all processes.

Figure 4-4: *Avoiding the test for waiting lists by storing the return address in a locked node. See also figure 3-4.*

do this on the transputer is shown in figure 4-4.

We have not yet implemented this technique as it may interfere with some experiments with different forms of evaluation. A clear example is compiling strict continuations, so that processes do not return after updating a node - i.e. reduce to root normal form -, but proceed with the evaluation of another function. We could store a continuation address in the node, similarly to storing a return address, but this may be more expensive than checking for a waiting list and jumping to the continuation code directly. It is not yet entirely clear if and how a combination should be realised. This largely depends on the need for different forms of evaluation. Meanwhile, we check waiting lists explicitly.

Note that we have not yet considered the overheads of starting and stopping processes. They should be kept to a minimum, but nonetheless they may become dominant when many small processes are created, or when certain dependencies exist between producing and consuming processes. If so, we think this should primarily be solved by increasing grain size. It remains to be seen whether this is feasible in all circumstances.

4.5. Supporting context switches

Clean requires fairness, so we need context switching. The transputer hardware automatically provides this, but only at the execution of unconditional jump instructions. As a result we must ensure that jump instructions are encountered regularly.

In contrast to C implementations, we have not used the transputer *call* instruction to realise subroutine calls. Instead we generate code to store the return address explicitly followed by a jump instruction. This makes subroutine calls more expensive, but it improves context switching capabilities. We cannot use a similar construct to implement *jsr_eval* instructions, because no context switch may occur between calling and locking a node. In this case, additional jump instructions within the called function will introduce the necessary context switching points.

Table 4-3: *The combined overheads of locking and context switching support.*

	without support for parallelism	with support for parallelism	overhead
nfib 30	11.7 sec.	12.2 sec.	4.3 %
tak 24 16 8	11.0 sec.	11.4 sec.	3.6 %
queens	42.9 sec.	43.9 sec.	2.3 %
reverse	62.0 sec.	64.2 sec.	3.5 %
fast fourier	12.7 sec.	12.9 sec.	1.6 %
mandelbrot	139.9 sec.	141.5 sec.	1.1 %
raytrace	315.9 sec.	321.0 sec.	1.6 %
sieve	18.3 sec.	19.4 sec.	6.0 %
quicksort	4.8 sec.	4.9 sec.	2.1 %

Limiting context switches to well-known places only is invaluable for a fast parallel implementation. It not only allows efficient manipulation of possibly shared data structures - such as nodes and the transputer evaluation stack -, but it also makes it possible to pass parameters in registers, just as in sequential implementations. To allow this we need to be able to tell which registers contain pointers during garbage collections. These may occur when a process is descheduled: either after executing a jump instruction, or after suspending explicitly. We provide pointer information by storing it in a special register just before such a situation arises. Arbitrary context switching points would not have allowed this.

A better solution would be to include pointer information in the code. It should be placed near suspend instructions, and before each entry that is reachable by an unconditional jump. The garbage collector is then able to retrieve it by examining the instruction pointer of each descheduled process. This will decrease execution time slightly, but more importantly it allows an additional register to be used for passing parameters. We have not yet implemented this due to the relatively small gains we expect. Check table 4-3 to see the current overheads for supporting context switches and locking.

4.6. Performance measurements for sequential programs

In this section we will present some performance figures for sequential programs. In case of the transputer implementation of Concurrent Clean the figures below include overheads for supporting parallelism (stack checks, testing for waiting lists, etc.), although the programs themselves are purely sequential. The Concurrent Clean figures will be related to transputer implementations of other functional languages. It will become clear that our transputer implementation performs relatively well. But before we proceed with this, we will relate the transputer implementation to a sequential implementation of Concurrent Clean for the SPARC processor.

As our transputer implementation uses simulated registers one would expect a performance penalty. Compared to similar implementations of Clean on a SPARC processor the performance on the transputer is indeed rather disappointing (table 4-4). However, if we consider the relative performance of C programs and the performance of other functional languages on the transputer, it becomes clear that it is not the implementation that is slow, but the transputer hardware itself. This is partly due to the age of the transputer design. It originates from the early 80’s and much faster machines have become available since then. It is still being used, but more and more as a real-time processing system and not so much as a number cruncher.

It is rather hard to compare functional languages on the transputer in a sensible way. Only a few functional languages have been implemented on the transputer hardware, and little is known about their performance. The notorious nfib benchmark is the only one that has been tested for a number of languages in a fairly consistent way. It does give some information, but one cannot not draw general conclusions from it.

Table 4-4: *Performance of Clean and C on a single transputer running at 25 MHz, compared to performance on a SUN4 (SPARC) running at 32 MHz. The SUN4 version is between 3 and 6.4 times faster than the transputer version. The transputer figures for reverse and sieve are not entirely the same as in the previous tables. The reverse function has been run in a smaller heap (2 Mbyte) and the sieve is an optimised version that avoids unnecessary divisions.*

	Helios C T800	C SUN4	speed-up
nfib 30	8.5 sec.	1.5 sec.	× 5.7
	Clean T800	Clean SUN4	speed-up
nfib 30	12.2 sec.	1.9 sec.	× 6.4
nfib 26 with reals	3.5 sec.	0.7 sec.	× 5.0
tak 24 16 8	11.4 sec.	1.8 sec.	× 6.3
queens	43.9 sec.	10.7 sec.	× 4.1
fast fourier	12.9 sec.	3.5 sec.	× 3.7
reverse	66.9 sec.	15.0 sec.	× 4.5
sieve	8.8 sec.	2.9 sec.	× 3.0

Table 4-5: *The sequential nfib ratings for various functional language implementations on transputers.*

PAM (20 MHz T800)	1.3 Knfib/sec.
HDG (25 MHz T800)	27 Knfib/sec.
Clean (25 MHz T800)	221 Knfib/sec.
Clean on ZAPP (20 MHz T800)	223 Knfib/sec.

Above, we find some results for PAM (Loogen *et al.*, 1989), the HDG machine (Kingdon, Lester and Burn, 1991), our Clean implementation, and a partial implementation of Clean on the ZAPP architecture (Goldsmith, McBurney and Sleep, 1993). The results of PAM have been obtained with an interpreter. This shows the need for code generation. The HDG machine on the other hand macro-expands abstract machine code to transputer code. It is mostly known for its evaluation-transformer graph reduction model, which is used to introduce parallelism automatically. And finally, the ZAPP implementation employs true code generation. It provides a virtual tree architecture that supports a divide-and-conquer style of parallel programming only. The ZAPP architecture has not been designed specifically for the purpose of running Clean programs.

The figures above suggest that the ZAPP implementation is the fastest functional language implementation and that the HDG machine is remarkably slower than both implementations of Clean. This is not entirely the case. The HDG machine does not have stacks. Instead, it uses large nodes to store stack frames. This makes it relatively slow for functions such as nfib and tak that gain significantly from using stacks. For the queens program the differences are less dramatic, although still notable (see table 4.6). The

opposite is the case for the ZAPP implementation. Compared to our implementation it benefits slightly from having a merged stack to evaluate functions like nfib. This advantage disappears for other programs (see table 4-7).

Table 4-6: *Performance of Clean and the HDG machine on a single processor.*

	HDG	Clean
nfib 20	0.81 sec.	0.11 sec.
tak 18 12 6	5.22 sec.	0.30 sec.
queens (on a board of 6 by 6)	0.21 sec.	0.08 sec.

Table 4-7: *Performance of Clean and Clean on ZAPP for a single processor. The reverse program reverses a list of 3000 elements 3000 times. The matrix multiplication program uses lists of lists of reals to represent matrices. The 25 MHz machine is not quite 1.25 times faster than the 20 MHz machine, due to substantial overheads in accessing memory (5 cycles for a word).*

	ZAPP (20 MHz)	Clean (25 MHz)
nfib 30	12.1 sec.	12.2 sec.
reverse	143.0 sec.	66.9 sec.
queens (on a board of 10 by 10)	62.7 sec.	43.9 sec.
matrix multiplication (64 by 64)	5.8 sec.	3.9 sec.

Altogether this is a meagre set of tests. Only a few programs have been run. Of these, the matrix multiplication example is the most realistic one. In particular the HDG machine has only executed very small examples. We expect that its large node layout will introduce problems for more substantial tests.

The functional language SkelML does not figure in the nfib suite above, as no results have been presented for nfib, but it has been tested on more realistic programs (Bratvold, 1993, 1994). SkelML is a version of Standard (read strict) ML that incorporates a set of skeletons to introduce parallelism. For the transputer a compiler has been constructed that generates Occam. Some tests have been performed with a ray tracing program. Table 4-8 contains the results for a single processor. It is about 3.5 times slower than our implementation of Clean (which is lazy).

Table 4-8: *Performance of SkelML and Clean. SkelML generates Occam, which clearly does not automatically produce the most efficient transputer executables.*

	SkelML	Clean
raytrace simple scene	1.86 sec.	0.53 sec.
raytrace complex scene	8.12 sec.	2.31 sec.

Analogous to the ZAPP implementation, C compilers can take full advantage of having to manage merely one stack on the transputer for functions like nfib (table 4-9). For

other programs the differences are smaller. We will see in chapter 8 that matrix multiplication in Clean can be equally fast as C on the transputer. We should also note that the relatively slow execution of nfib in Clean is purely a transputer problem. On true register-based architectures Clean runs this benchmark about as fast as C (sometimes a bit slower and sometimes a little faster, depending on the machine).

Table 4-9: *The sequential nfib ratings for C on transputers. The C compilers can take full advantage of not having to manage multiple stacks. The Pact compiler places the stack in on-chip memory by default. This may increase execution speed for sequential programs, but it is not realistic when multiple processes are running on a single processor. As we can see below, the Helios compiler produces the best code by far, but we should take into account that the Pact compiler provides more flexible runtime constructs for creating parallel programs. This complicates the calling conventions of the Pact Compiler, which decreases performance, in particular for tiny recursive functions like nfib.*

Pact C (stack in on-chip memory)	267 Knfib/sec.
Pact C (stack in external memory)	183 Knfib/sec.
Helios C (stack in external memory)	315 Knfib/sec.

We have also run some tests with interleaved concurrency on a single processor. In general these versions ran only a little slower than the sequential versions, due to overheads for starting and stopping processes. These overheads are highly dependent on the number of processes and their interaction. For divide-and-conquer programs the overheads were typically no more than 5%. Only for the sieve programs these overheads resulted in a considerable slow-down (it ran twice as slow as a purely sequential sieve). Some programs could not be run in an interleaved manner due to shortage of memory: too many processes needed to be accommodated on a single processor. Unfortunately, we have not been able to relate these results to other implementations, as these do not report performance figures for interleaved execution (which they usually do not support).

4.7. Conclusion

We have shown that it is not extremely difficult to adapt a register-based sequential code generator to transputer hardware. Simulating registers does not introduce serious problems. In addition, it keeps the transputer implementation compatible with more common architectures.

Furthermore, the additional constructs that are needed to support parallel evaluation do not introduce a serious performance penalty. In particular the overheads of reallocating stacks are negligible. It is possible to further reduce the costs for locking and the costs for supporting context switches. Additionally, avoiding intermediate ABC machine code might improve stack management, at the cost of merging low level and high level implementation techniques. Nonetheless, our implementation already compares favourably to other implementations of functional languages. In the next chapter we will present performance figures for parallel programs.

5. Managing Distributed Graphs

In the previous chapter we have shown how to compile abstract machine code to concrete machine code. The generated programs specify graph reduction in detail, but they do not realise graph copying, nor garbage collection. To complete graph rewriting, these complex features need to be incorporated in a runtime system. This chapter will show how to accomplish this.

In section 5.2 we will determine which data structures are needed to keep track of distributed graphs. Section 5.3 deals with transmission of graphs. It focuses on efficiency, feasibility, and reliability. In addition it will consider copying of work. In section 5.4 we will examine distributed garbage collection. The Clean implementation employs a combined approach. It uses a copying garbage collector for each processor heap, and a weighted reference counting scheme for the inter-processor references. Cycles and speculative parallelism complicate matters. We have not yet solved these problems, but we will show how they might be tackled. Section 5.5 will conclude this chapter with some parallel performance figures. This demonstrates the effectiveness of our graph management techniques. The Concurrent Clean implementation compares favourably to others, which are often more restrictive. The tests will also reveal some efficiency problems. These will be treated in the remaining chapters.

5.1. Introduction

Clean employs graph rewriting for its implementation. In a sequential system function nodes are evaluated by a single process according to the functional strategy. After a particular function node has been chosen for reduction, it will eventually be overwritten by its result[5]. In between, during reduction, new nodes may be constructed and other function nodes will be reduced if necessary.

In a parallel system with distributed memory the graphs are scattered over the network. Reduction then takes place at different processors simultaneously, possibly by multiple processes on the same processor. Locking of nodes prevents reduction of the same shared node by multiple processes. Sometimes the reduction of a function requires access to a part of a graph that is stored at another processor. If this is the case, this part needs to be transported to the function that needs it, possibly after evaluation.

[5] It might be possible to avoid updating nodes by taking into account uniqueness information, but this has not yet been investigated.

Garbage collection and transmission of graphs both play an essential role in a finite parallel distributed system. They complement graph reduction as described above. Still, transmission of graphs and garbage collection remain rather abstract notions within the abstract machine code, although it does specify graph reduction in detail. The ABC machine code tells how to reduce functions - that is, how to construct graphs -, when to reduce a function and - related to this - when to request a remote graph. Yet, instructions that indicate the need for a certain graph at another processor can hardly be regarded to represent a basic abstract reduction step. At the same time garbage collection remains completely hidden. The abstract machine assumes that garbage collection and transmission of graphs exist, and it tells what they should do, but it does not clarify how they should be realised.

Solving both problems is far from trivial. If one considers garbage collection, one may choose from a number of techniques, varying from on-the-fly collectors, to stop-and-collect techniques. Each has its advantages. Similar problems exist for transmitting graphs. How do we arrange transmission? What protocol do we need? What happens when a graph is requested that for some reason cannot be transmitted directly?

A complicating factor with respect to garbage collection is that Clean allows cyclic graphs and speculative parallelism. Dealing with cycles in a system with distributed memory is known as a hard problem. Additionally, speculative parallelism allows existence of processes that produce results that are not necessarily needed. If at some point in time it becomes clear that such a process produces unneeded results - it produces garbage -, this process should be stopped by the garbage collector. Detecting such a garbage process is a delicate problem in itself, but additionally, it may be running too fast for the garbage collector to catch it.

As we have explained in chapter one, Clean employs a lazy graph copying mechanism. This allows multiple nodes to be transmitted simultaneously so that small messages are avoided. Small messages give rise to relatively high overheads, so lazy graph copying reduces communication costs, which is very important in a distributed system. To realise such a graph copying mechanism we need to copy (or move) nodes locally into a single message. Thus, the nodes of a graph are gathered (packed) before transmission. This packing will be referred to as graph copying; building a local duplicate of a graph (in some appropriate format). If we move graphs, the original will be discarded after copying. Thus, it is important to make a distinction between (local) graph copying and transmitting graphs to other processors. The latter requires the former.

It is here that garbage collection and transmission of graphs come together. Garbage collection requires a graph copying mechanism as well. This is most apparent for garbage collectors in sequential systems. In contrast to building a message, the garbage collector uses it to move and compact structures that are in use. This not only applies to a copying garbage collector, which will copy live data to a new heap and discard the old one, but also to mark-sweep garbage collectors, which will gradually copy live data to one end of the heap, thereby overwriting the old structures. Consequently, graph copying and copying garbage collection for a sequential system are almost synonymous.

This chapter will treat garbage collection and transmission of graphs simultaneously, partly due to this strong relation. However, we will not focus on the graph copying

algorithm itself, which is fairly straightforward, akin to a common copying garbage collection mechanism. Instead we will show how graph copying can be used to realise distributed garbage collection and transmission of graphs. We will demonstrate which communication protocols and which data structures we employed to manage distributed graphs. The main emphasis will be on implementation issues such as efficiency, reliability and feasibility.

5.2. Representing references to remote graphs

To manage distributed graphs we need a few additional data structures. Basically, we require a way to represent a references to remote graphs. This section will explain how this has been provided in the Clean system.

5.2.1. Channel nodes

The parallel ABC machine uses special function nodes - channel nodes - to refer to graph at other processors. Such a channel node has the same semantics as an ordinary indirection node. Evaluation is equivalent to evaluating the remote graph itself if it were stored at the current processor. As a consequence, every channel node that is being reduced should eventually be overwritten by the root normal form of the remote graph that it refers to.

To achieve this behaviour, evaluation of a channel node will result in a series of operations. First, after locking the channel node, a request message will be sent to the processor that contains the referred graph. If the graph is in root normal form it will be returned immediately. The channel node will then be overwritten by the evaluated root node. And finally, the processes that are suspended on the channel node will be released.

However, it may be the case that the requested graph has not been reduced yet. If a process is already reducing it, we simply administer the request by inserting a special answering function in the waiting list of the remote graph. Otherwise, we start a new process on the graph first. The functional strategy guarantees that a root normal form will be reached if it exists. After evaluation (to Root Normal Form), the reducer will release the waiting list of the remote graph by evaluating all functions in it. Thus, all requests will be answered, and the channel node will eventually be updated, analogous to the previous paragraph.

In practice, handling requests can be slightly more complicated sometimes. The Clean runtime system makes extensive use of asynchronous communications and adaptive routing to avoid unnecessary sequentialisation. As a result, it is possible that a request arrives ahead of the graph itself. In such cases we create place-holders (empty nodes) to keep track of requests. The graph itself - i.e. its root node - will then be placed in this empty node when it arrives.

5.2.2. Indirection tables

Keeping track of remote graphs is not trivial, because the addresses of nodes may vary, due to compacting garbage collections, as we will see below. To cope with this we will use an indirection table. Each processor contains such a table and its entries refers to the locally stored graphs that are accessible from outside. If the garbage collector adjust the pointers in

the indirection table the indices provide fixed node addresses (global addresses) that can be used in references. Similar approaches have been used in various other implementations (the PABC simulator, the HDG machine, and the GUM implementation of Haskell).

Channel nodes contain global addresses that consist of two parts: the processor number and the index of the indirection table entry that refers to the graph. Figure 5-1 depicts the relation between channel nodes and indirection tables in detail.

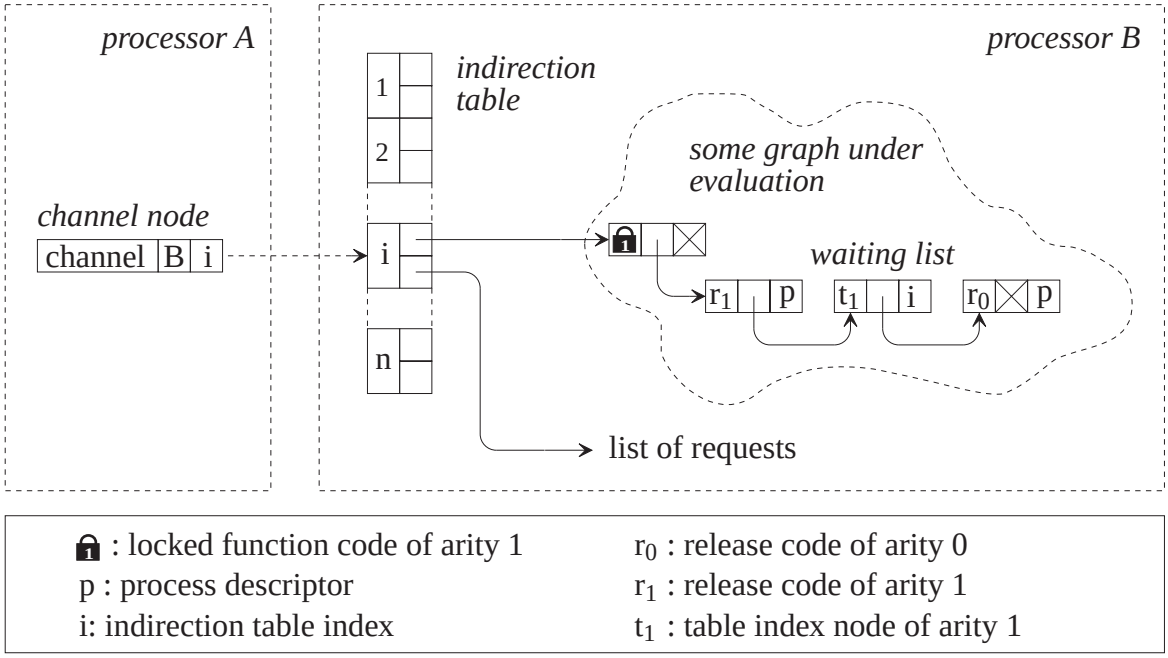


Figure 5-1: *The relation between channel nodes and indirection tables. The Clean runtime system stores an indirection table in the heap of each processor. If it turns out to be too small it is reallocated in the same way that stacks are. The table may shrink as well, but not beyond the highest entry that is in use. Waiting processes can be placed in the waiting lists of both the channel node and the remote graph (the channel node above does not yet have any, as it will only contain a waiting list field after it has been locked, in which case it is actually not a channel node anymore, but a locked node). Requests are stored in the indirection table. It acts as a place-holder when a request arrives ahead of the graph itself. The waiting list of the globally known graph does not only hold references to waiting processes (i.e. local ones), but also the index of the corresponding table entry, so that requests can be retrieved - and answered- by the reducer that updates the graph.. To accomplish this, the reducer merely has to evaluate the evaluation code of the waiting list element that contains the index.*

On creation of a channel node to some processor, we need to reserve a unique index in the corresponding indirection table. One way to deal with this, is to shape the indirection table as a hashing table. On creation of a channel node one can simply generate a new hashing key locally. One might take a combination of the processor address and some locally unique number, so that the resulting key is globally unique. The hashing table will

map this key to a single table entry. The advantage is that no delays are introduced on obtaining an address. The disadvantage however, is that managing the indirection table becomes more complex. Not only are additional structures needed to accommodate multiple graphs in a single hash table entry, but this method also allows the use of a key for which no corresponding physical table entry has been created yet. This can be problematic if at some point in time status information needs to be stored in a table entry during garbage collections, for instance to detect communication errors (see the subsection on deadlock-free protocols). Dealing with such problems in another way may be less efficient, and it can be a nuisance, especially in prototype implementations, because it complicates communication protocols.

We have adopted another solution that simply uses buffered streams of allocated indexes coming from different processors. Allocating an index can be achieved by taking an index from the appropriate stream. In general, this avoids delays. Such a stream has the additional advantage that it is able to carry remote load information. This solution turned out to be sufficient for the programs we tested. However, it might be necessary to switch to using hashing tables if obtaining global addresses proves to be a bottleneck nonetheless.

5.2.3. Tracking of duplicates

Channel nodes can be duplicated and transmitted to other processors. It is possible that different channel nodes to the same graph end up at the same processor. The first time that one of these references is evaluated the graph will be copied to the processor that holds the channel node. However, evaluation of the other reference will result in another duplicate at the same processor. If possible, this should be avoided. However, we will see below that dealing with this problem can add too much complexity to the implementation.

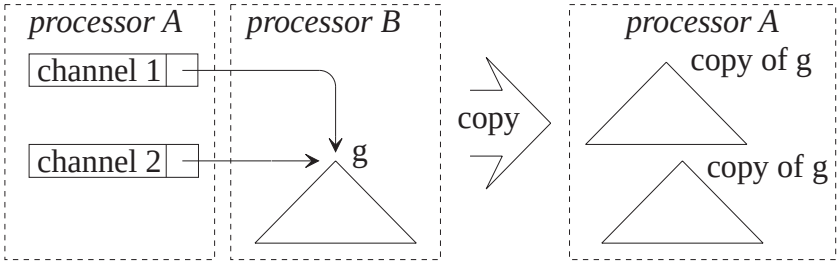


Figure 5-2: *unwanted introduction of duplicates*

We should note that this problem does not occur if one merely shares a remote expression. In this case one would share the reference to the remote object (the channel node) and not the remote object itself. To obtain situation above, one would either have to copy a graph that incorporates two different deferred nodes that share some sub-graph, or one would have to create two processes that share a deferred graph and start them up at the same processor. Clearly some effort is required to obtain the situation of figure 5-2.

One way to deal with this, is to maintain sharing of graphs in all circumstances, so that duplication of graphs is avoided altogether. Graphs will not be copied, but moved.

Different references to the same graph may exist, but if the graph moves all references will be redirected to its new location. On the one hand this avoids needless duplications, but on the other hand it also introduces needless retransmissions if graphs are needed at several processors. So instead of not detecting some locally cached graph, we have obtained a severely restricted form of caching, which in some cases equals no caching at all (see also the subsection below on copying work). This can have a serious impact on performance, so we will not consider this here. In contrast, we will focus on detecting references to different instances of the same graph. We will discuss the possibility of tracking local duplicates in order to avoid unnecessary additional duplication.

The GUM implementation of Haskell uses two hashing tables to accomplish this in certain situations (Hammond *et al*., 1995). The first maps local address to global addresses. This is used to ensure that every graph has at most one global address, so that references to the same graph are easily recognised. The second table maps global addresses to local ones. This identifies locally available copies of the remote graph. On evaluation of a channel node one first checks if the graph is locally available already.

There are some problems with such a solution. Depending on its intended scope it can imply notable extra costs, and even then it may not be able to track all duplicates. Consider the case that some graph *g* at processor *B* gets copied to processor *A* and that a channel node at *A* refers to a sub-graph *s* of *g* at processor *B* (see figure 5-3). Transmission of *s* to *A* results in a partial duplicate of *g*. If we use hashing tables to detect this form of sharing as well, transmission of graphs will become considerably more costly. During the original transmission of *g* to *A* we would have had to check the entire original graph at *B* for possible remote references before sending. The detected global addresses should have been included in the message and used to update the hashing tables at *A*, enabling it to detect local availability of *s* later. This is highly undesirable, because without such mechanisms graph copying already can be a bottleneck (see chapter 7). Note that we have to include all global addresses in the message, not just the ones from *A*, because we normally do not know where references come from (as keeping track of this would be expensive in a distributed system and irrelevant for ordinary reduction).

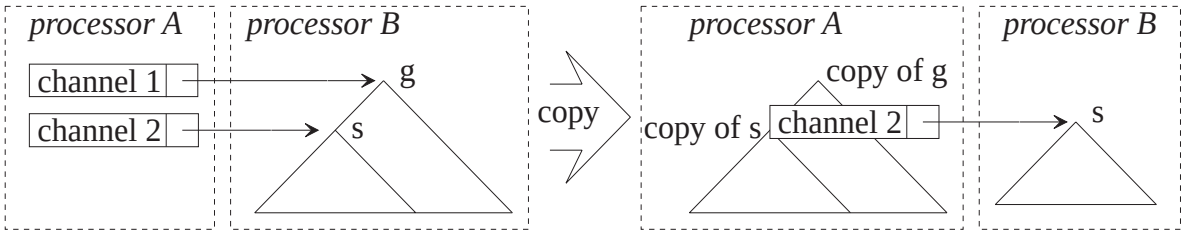


Figure 5-3: *Subsequent evaluation of channel 1 and channel 2 will result in duplication of s at A.*

Still, all this extra work is not sufficient. A channel node may possibly not refer to a shared sub-graph directly. Reconsider the example above and reverse the order in which channel nodes are evaluated. At some point in time, processor *A* will then have received a

sub-graph *s* from processor *B*. It additionally contains a channel node that refers to *g*, which might be a single non-shared node at processor *B* that refers to *s* (see figure 5-4). For instance, the root of *g* may be a selector function on *s*, or more obscurely, the result of such a selector function: even if the original selector node is overwritten by a node from *s*, its global address will differ from the one in *s*, which possibly does not have any global address at all. If processor *A* tries to map the global address of *g* to a local one this will not succeed, because it does not have a local equivalent of *g* already. To deal with this - i.e. to avoid sending *s* for a second time -, processor *B* would at least have to keep track of the destination to which it has copied any of its nodes earlier. In addition it would have to know which of the copies still exist. This would be very expensive.

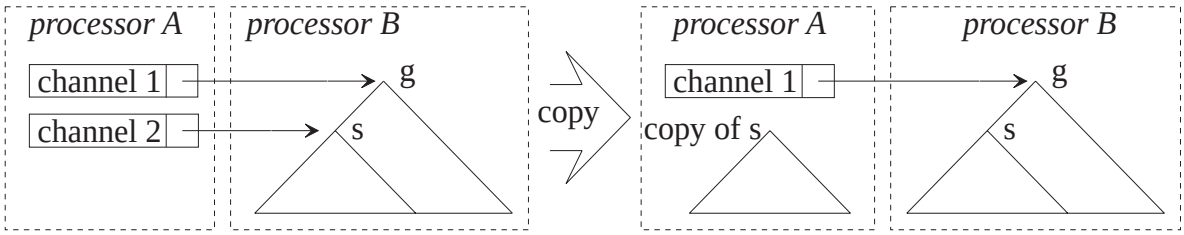


Figure 5-4: *Subsequent evaluation of channel 2 and channel 1 will result in duplication of s at A*

The proposed hashing mechanism may be insufficiently powerful, but the main question is, whether the cases it *can* detect justify its implementation. This is unclear. In any case, it has serious problems in a few important cases. Firstly, during the creation of two processes on a shared argument it will not automatically avoid copying the argument twice, because then the initiative for transmission is taken by the sender and not by the receiver. The hashing method will only become effective after requesting additional parts of an argument (if part of the argument was deferred). In addition, as has been pointed out above, it has serious problems when two graphs - say *g* and *g'* - partly overlap as shown in figure 5-3 and 5-4. This form of sharing is easily introduced in lazy functional languages, for instance, by passing a single argument to two functions. It is questionable whether the remaining detectable cases constitute a major part of the duplication problem in practice.

And even if they do, it might be better to allow the programmer to deal with this so that runtime overheads are avoided. For instance, instead of starting up two different remote processes with the same argument, one rather might have started up merely one. This process could then start up additional internal processes which all share a single channel node to the argument. Alternatively, one could also transmit the argument before starting up several processes on it (simply by starting an identity function on the argument at the correct processor and subsequently starting the processes on the result). Alternatively, using random process allocations in large networks will make it less likely that different references to the same graph end up at the same processor. And finally, it might be possible to conveniently deal with sharing and duplication with suitable language constructs and (profiling) tools.

In any case, the best way to solve this problem is still unknown. We have not yet acquired enough experience with parallel functional programming to establish the need for tracking (certain) duplicates. Nor do we know whether programmers - or runtime systems - can adequately deal with this problem. For practical reasons, we have chosen to refrain from attempting to implement - and test - hashing techniques to partially track duplication of graphs. This would have introduced too much work, while the example programs in this thesis do not suffer from the problems depicted above. We only try to maintain sharing within the graphs that are transmitted in a single message. This can be done in a relatively cheap way, using forwarding addresses during graph copying. In the next section we will have a closer look at this.

5.3. Transmission of graphs

Transferring a graph to another processor involves several phases. First of all, we must pack it in a message. Next, we can transmit it to another processor, possibly after ensuring enough memory is available. Doing so, one has to be careful not to introduce deadlocks. And finally, after the copy has arrived, it needs to be unpacked, which mainly involves adjusting pointers. We will examine these phases in more detail below. In addition we will discuss the possibility of avoiding copying of work, as this is closely related to the way that graphs are transmitted to other processors. And finally we will focus on realising deadlock-free protocols.

5.3.1. Packing a graph

To pack a graph, the Concurrent Clean graph copying mechanism walks over the original graph and copies all nodes it encounters in a contiguous area in the heap. It uses marks and forwarding addresses to detect and maintain local sharing and local cycles. Doing so, the original graph gets corrupted, but it can be restored afterwards, by traversing the copy.

Copying will stop at deferred nodes. A new reference (a channel node) will be created to such a node, which replaces the copy. The effect is, that copying gets postponed at deferred nodes. It will automatically be resumed later if a process tries to de-reference - i.e. evaluate - the new channel node (see the previous section). Note that any copying mechanism should at least copy the root node of a graph that has been reduced to root normal form, because the goal of transmitting a graph in root normal form is to update a channel node with it.

In addition to deferred nodes, copying will always stop at channel nodes. More precisely, copying will stop after copying a channel node: the channel node itself gets copied, but it will not be de-referenced. Thus, a new reference to the remote graph is automatically created. Clearly, this is the best thing to do, as there is no reason why a remote graph should be copied to the current processor only to be sent away to another one (except perhaps for maintaining sharing at extremely high costs).

The partial duplicate that is obtained this way can almost be put in a single message and sent to another processor. It only lacks relocation information. To be able to adjust argument pointers we include the address of the copied root node. Descriptor fields are no problem, as these contain indices in a (standard) table that is stored on every processor. A

constructor is represented by the same index throughout the network. Code pointers on the other hand may vary and therefore, they are replaced by their corresponding descriptor index (see also figure 3-6). This also allows the use of heterogeneous networks.

It would be possible to compress the copy before transmission (see also van Groningen, 1992). We will not do this, as the transmission costs are usually not a bottleneck on the transputer. In contrast, packing itself can be more of a problem, as we will see in chapter 7. This will only become worse when using compression techniques.

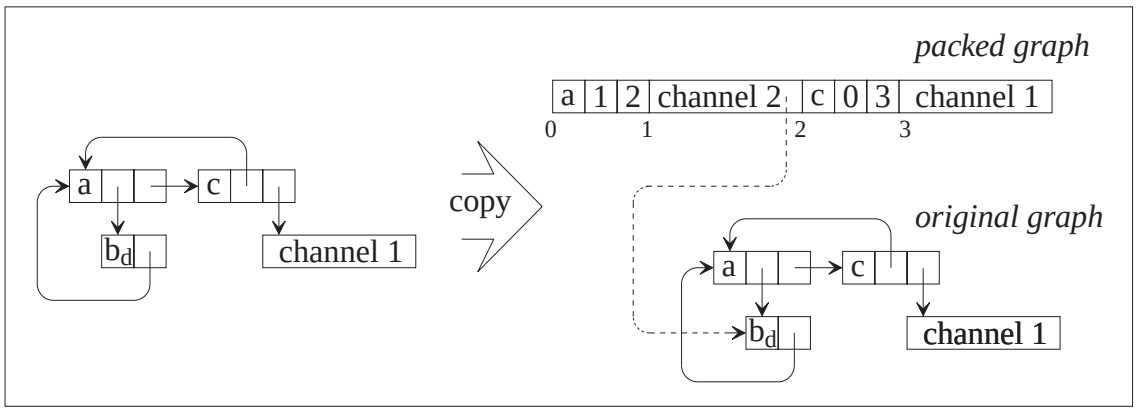


Figure 5-5: *Packing a graph rooted by node ‘a’. The node with symbol ‘b’ is deferred. The top cycle is broken up here. In contrast, the cycle over ‘a’ and ‘c’ is maintained.*

5.3.2. Transmission

Messages are always created in the heap, which eliminates the need to implement separate message buffers (of arbitrary size). They are constructed from standard nodes (constructors), so that reducers conveniently can carry around messages on their stack without any danger of messages being destroyed by the garbage collector. This also allows sharing of (parts of) messages, which can be employed to send large messages to several processors without having to duplicate them physically in the heap.

Creating messages in the heap has a downside as well. Garbage collections will move heap objects. Consequently, messages are subject to relocation. This may happen while they are being sent by the transputer link hardware, which operates independently of the CPU. For this reason, the links cannot simply read - or write - data directly from the heap during a garbage collection. Therefore, the router (see chapter 2) always copies messages in its own buffers before sending. If necessary, this can be avoided, during the time between two garbage collections, but only at the expense of extra checks within the router software.

To transmit a message, it merely needs to be appended to a special list of outgoing messages in the heap. The router continuously checks this list and picks out messages for transmission (in a fair manner). It guarantees arrival, so after putting a message in the message list, one can safely proceed without waiting - or checking - for an acknowledgement. Consequently, communication is asynchronous as far as reducers are concerned. Additionally, the router splits up large messages into packets, to which it appends the position in the original message. This allows the receiver to reconstruct the original message from all parts.

To facilitate unpacking of graph messages, we guarantee that the constituent packets will be stored in the correct order in a contiguous (relocatable) area of the destination heap. To achieve this, we need to allocate this area before storing the first packet that arrives. This does not have to be the first packet of the original message, as packets may arrive out of order. For this reason each packet includes the size of the whole message. A drawback is, that we cannot overlap packing and transmission, unless we split both tasks up, which might lead to some additional loss of sharing.

We use different allocation protocols depending on the size of a message. For small messages, we assume that enough free space is available in the destination heap. We just transmit the message and allocate memory at the moment the first packet arrives. Most of the time this will succeed, keeping delays to a minimum. However, if it fails, the whole message is discarded and a new one must be transmitted. This increases the transmission costs occasionally, but for small message this will not be too serious. This protocol requires that the source processor keeps a copy of the message for as long as the destination process has not properly stored it. Retransmission and acknowledgement messages are handled by the processes of the underlying runtime system. For reducers, transmission appears to be asynchronous.

Large messages on the other hand, will not be transmitted until memory has been allocated at the destination. This avoids costly retransmissions. On the other hand, delays will be larger because the destination has to grant permission for transmission. However, the runtime system handles the control messages that are needed to achieve this, so that reducers can safely proceed after submitting a message.

5.3.3. Arrival

The router never queues incoming packets to be processed later, as this may give rise to buffering problems (see also the section on deadlock-free protocols below). It rather processes packets directly upon arrival. It has associated a packet-handling routine with each kind of packet. This will be executed as soon as a packet arrives. To avoid deadlocks, the runtime system only employs routines that never fail and immediately free the router buffer (see also section 5.3.5).

After all components of a messages have arrived, the contained graph will be unpacked. This will take place within the memory space of the message itself, so we do not need additional memory. Argument pointers are adjusted according to the difference between the original address of the root node and the new one. The descriptors of constructors do not need to be changed. Conversely, the descriptors that are stored in the packed function nodes are replaced by their corresponding code addresses. These can be found in the local copy of descriptor table (see figure 3-6).

Transmission of the graph has now completed. Note however that parts of the original graph at the source processor may still exist. If they are shared by other processes the garbage collector will not remove them. If so, we have actually duplicated (part of) the graph.

5.3.4. Avoiding duplication of work

This leads us to an important problem, namely duplication of work, represented by function nodes. This is especially hard, because there is no general way to predict the costs of evaluating a function node. Potentially, functions represent an infinite amount of work. Copying function nodes might quickly lead to vast duplication of work. This can be disastrous for performance.

For comparison, copying data - represented by constructors - is less of a problem, because in many cases it will be useful to keep copies of data locally available. The use of caches in (virtual) shared memory systems is based on the same assumption, using the property that normal forms are unchanging to ensure cache coherency. It will only affect memory usage and not processor load as well, like duplicating work will. One will encounter difficulties only when memory is low. With respect to possible techniques to reshare expressions, note that equality of data - normal forms - is fundamentally easier to detect than equality of functions.

In contrast, *moving* data (as opposed to copying) will give rise to unacceptable overheads if it is needed simultaneously by several processors (all pulling at it). Additionally it will seriously complicate the reduction mechanism, as one cannot be sure that an evaluated argument will stay locally available for important operations like pattern matching.

As a result, data should in principle be copied as opposed to functions (and possibly processes), which should be moved. If we want data objects to stay unique, they should not be shared, but passed explicitly between functions.

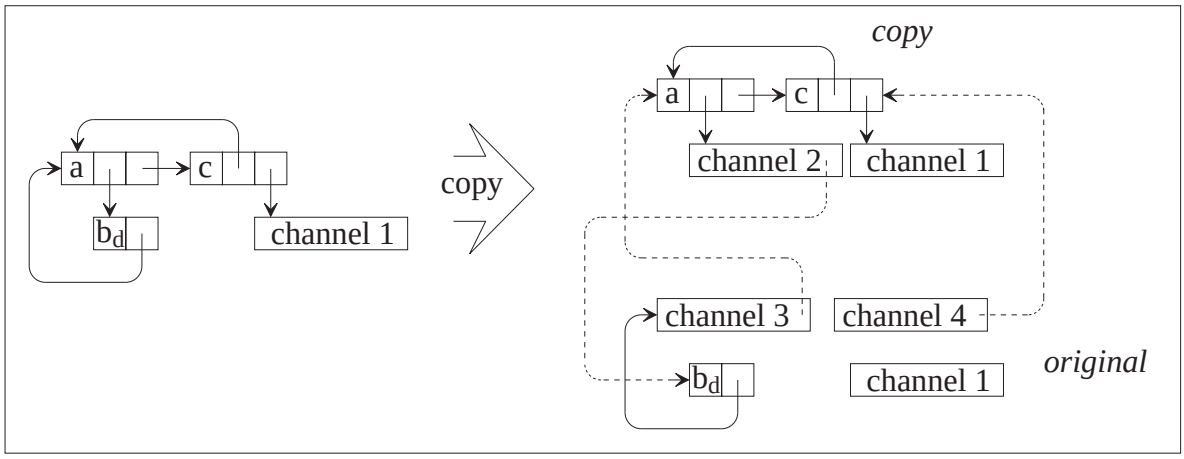


Figure 5-6: *Moving work instead of duplicating it: if node 'a' and 'c' are function nodes, the originals will be replaced by channel nodes to their copies (see also figure 5-5).*

To realise a graph transmission mechanism that truly moves function nodes we need to do extra work. The implementation of Haskell on GUM for instance (Hammond *et al*, 1995), replaces all function nodes of the original graph by references to the corresponding copies. The first stage of this algorithm is virtually the same as the one above. Hereafter, it constructs a reply message at the destination processor that includes the new addresses of copied function nodes. After returning this message to the source processor, the original

function nodes are updated by channel nodes that refer to their copies. To avoid reduction of the original function nodes while they are being moved, they remain locked during this time.

To decrease delays on evaluating a node that is being moved one might allocate references to the new positions on beforehand: one may update the original function nodes with references to remote place-holders that will eventually be filled with the copy. In this way, a process will not merely block on a moving node, but send a request for it as well (which might even overtake the node itself).

Unfortunately, this moving technique has some practical disadvantages. First of all, the time it takes to complete a successful transmission might become rather high in large systems. In general, communication costs will increase because a - possibly large - reply message needs to be sent. Secondly, handling large graphs becomes more complex, because now not only space has to be reserved for the graph, but also for the reply message, which may have a considerable size as well (see also the subsection on deadlock-free protocols). Thirdly, moving a function node repeatedly will leave a trail of channel nodes behind. One will have to traverse these in order to get to the result of the function. Such chains have indeed been observed for some programs in the GUM implementation of Haskell. And finally, the moving algorithm is not able to determine which function nodes it *needs* to replace by a reference to the corresponding copy. It simply replaces all, because it lacks sharing information. Apart from the - possibly pointless - replacement overheads themselves, this introduces two additional drawbacks. Firstly, adding references needlessly, will result in superfluous overheads during subsequent garbage collections, which will remove these references (as the garbage collector does have enough sharing information). Secondly, as we can see in figure 5-8 (reconsider also figure 3-3), nodes may not be large enough to hold a reference. As a result, updating a graph may increase its size considerably, leading to all sorts of memory management problems.

As a consequence we will use a different way to avoid the dangers of copying work. Instead of moving function nodes, we will never copy them implicitly. One can only copy work by annotating it, so that it is only duplicated explicitly. The next chapter will focus on this in detail. It will show that this solution does not lead to an awkward style of programming but that programs more clearly reveal what will be computed where. It will provide distributed lazy normal form processing.

5.3.5. Realising deadlock-free Protocols

The routing mechanism that has been presented in chapter 2 prevents deadlocks provided that messages are consumed within a finite amount of time at the destination. Other routers inevitably have a similar requirement, which protocols built on top have to fulfil in order to provide deadlock-free communications. For this reason, the Clean runtime system immediately stores a packet on arrival, processes it in some way, or discards it (e.g. during garbage collections), so that the router buffers are freed as soon as possible.

A delicate problem is imposed by messages that require some answer to be returned. The reply message cannot always be created in the heap, because a processor may temporarily have run out of heap space (during garbage collections). Likewise, one cannot simply postpone creation of such a message until more convenient times, because this

either blocks the routing mechanism for a considerable time, or it requires administration of the request, which consumes memory as well. On the other hand one cannot use the buffer space of the incoming message to construct the answer, as this could introduce deadlocks. To deal with this, either the routing mechanism has to be adapted - for instance by increasing the number of buffer classes -, or one has introduce additional constructs at a higher level that allow freeing a buffer directly on arrival of a message.

Extending the routing mechanism seems an elegant way to provide safe reply messages. It has the advantage that higher level protocols remain simple. However, the proper way to do this depends on the characteristics of these higher level protocols (such as the size of reply messages and the number of times a message may 'bounce' back and forth). Especially in a prototype implementation, these are likely to change. In addition, routing mechanisms typically exploit certain properties of the network topology. The Clean router for instance, allocates buffer space according to the network diameter. Such an approach is rather static, in the sense that it cannot not adapt its use of resources to runtime demands. And finally, changing a router requires careful reconsideration of the algorithm it uses. These are often complex and hard to adjust. Hardware routers may not be expandable at all. This limits portability. Therefore, we will stay with a basic routing mechanism and examine if we can devise a safe protocol on top of it.

One way to achieve this would be to allocate reply messages in a separate memory space. Time-outs could be used to detect failure due to lack of memory. The problem with such a solution is that it complicates the end-to-end protocol considerably. One would have to deal with needless retransmissions, and tuning the system to a particular network topology. These problems are essentially the same as those in a system that contains unreliable communication links. As transputer networks - and many other parallel machines - commonly do not belong to this category, it seems odd to adopt such a solution.

A better way would be to limit the number of requests a processor can submit to another one at any moment in time. If the number of unanswered requests reaches a certain point, new requests will be queued at the source processor, or sent to a different processor if this is possible. The destination should have allocated enough memory in advance to be able to handle any request up to this allowance. Note that reply messages should have a limited size as well in order to make this possible.

A small allowance might introduce too much synchronisation between two processors. To deal with this, it will need to be increased, requiring the allocation of additional memory for constructing reply messages. This can be done at runtime, which is a considerable advantage compared to extending the underlying routing mechanisms. Depending on the memory use, a processor may alter the allowances of others. This will also help in balancing the processor load.

In a way, the Clean runtime system already incorporates a simple form of a self-regulating allowance. If we allocate an entry in the indirection table, we automatically increase the allowance for handling certain messages. Each entry provides some (tiny) amount of space for constructing, postponing, or even avoiding reply messages. Note that this requires the physical existence of a table entry in certain cases, which makes the use of locally generated hashing keys to represent indirection table indices less attractive (see section 5.2.2).

We have not yet adopted any complete solution for practical reasons. In our prototype implementation protocols are likely to change, while future communication hardware is all but established. At the moment we use a dedicated message buffer for allocating reply messages. This has turned out to be sufficient for most tests. Only in a few cases, after memory became scarce, the number of reply messages increased to such extent that the buffer became depleted.

5.4. Garbage collection

As the previous section has pointed out, many problems are related to memory management. They do not figure in ideal systems that have an unlimited amount of memory. Real systems with finite memory need some form of garbage collection that reclaims unused space. This complements graph transmission and graph creation.

5.4.1. Sequential garbage collection

Copying garbage collectors (see Minsky, 1963) are commonly used in sequential systems. They are known for their efficiency if relatively little memory is in use. Additionally, they automatically compact the heap so that fragmentation is avoided and fast memory allocation is sustained. The main drawback is that only half of the available memory can be allocated.

Mark-scan collectors (see Cohen 1981) do not have this disadvantage, but they are less efficient when little memory is used, as they need to scan the entire memory space. Therefore, the sequential Clean implementation combines both techniques to track live data (Sansom, 1991). It uses a copying collector when memory demands are low, and switches to a compacting mark-scan (mark-sweep) collector (van Groningen, 1993) when memory usage surpasses a certain threshold.

Reference counting mechanisms are less appropriate for sequential implementations. This has several reasons. First of all they do not automatically compact the heap, which may lead to fragmentation if nodes have various sizes. Secondly, they tend to increase node size, as reference counts need to be accommodated in each node (except perhaps for nodes for which the reference count is known at compile time. See also the next chapter on uniqueness). In particular, this is a problem for small nodes. Thirdly, reference counts need to be maintained always, even in the face of abundant memory, giving rise to unnecessary computational overheads in this case (note however that reference counting can be cheaper if memory is scarce). But most importantly, reference counting cannot reclaim cyclic structures.

5.4.2. Distributed garbage collection

In an architecture with distributed memory though, collectors that *track* live data (tracking collectors, such as copying collectors and mark-scan collectors) are less valuable. First of all, they do not have an apparent distributed equivalent. To be effective they need to construct a global overview of the distributed system. This may take considerable time due to communication overheads. Meanwhile, garbage will persist. This can introduce considerable delays at processors that do not have any free memory left. The global view

may be blurred as well. Unless one forces all processors to a full stop - which can be extremely costly -, one can only compute an approximation. Secondly, in a distributed memory machine, there is no clear relation between the total amount of garbage and the invocation of a garbage collection. Some processor heaps may be full, but many others may not contain much garbage. This contrasts sharply with tracking garbage collections in sequential implementations, which will only be applied when there is absolutely no space left and a reasonable amount of garbage is likely to exist. This makes the efficiency of copying and mark-scan collectors hard to predict in a distributed environment.

For comparison, reference counting operates on-the-fly by default, so that processors do not need to be stopped. It removes garbage as soon as it is comes into existence - which avoids delays -, and it remains idle otherwise. The costs do not depend on quantities that are indirectly related to the amount of garbage, such as the proportion of the heap that is in use, the total amount of heap space, or the frequency of garbage collections. The overheads solely depend on the amount of garbage that is produced and up to a certain point they are localised to the appropriate areas as well (only nodes that are reachable from the detached one are affected).

Furthermore, one can avoid many of the disadvantages of reference counting by employing it for inter-processor references only (see also Kingdon *et al.* 1991). For each processor this will determine which globally accessible nodes are garbage. Locally, a traditional tracking garbage collector can be used to remove all nodes that are not reachable from outside (through one of the live global nodes). This collector will basically be invoked when the local heap fills up, analogous to single-processor implementations. In turn, this will indicate which references to other processors are garbage. This is related to lazy garbage collection, which avoids a recursion stack to update reference counts (Glaser and Thompson, 1985). In this way we retain the advantages of copying and mark-scan collectors for most nodes (compaction of each local heap, small node size, reclamation of local cyclic structures).

On the other hand, this hybrid solution does reintroduce some of the delays that are typical for tracking collectors. To deal with this problem, it may be necessary to decrease delays by introducing additional local garbage collections for advancing reference count information in time. One sometimes needs local garbage collections on lightly loaded processors to allow removal of garbage on heavily loaded ones.

Nonetheless, the hybrid solution introduces less overheads than a global tracking mechanism. The latter *always* introduces garbage collection overheads on lightly loaded processors, whereas the former can limit the number of extra collections by introducing them only when ordinary collections could not free enough memory on certain processors (that lightly loaded processors have references to). In general, delays are less serious for the hybrid garbage collector, as they are not introduced at the top level of the distributed garbage collection, but at each node that gets detached. All in all, the combined solution is better suited for distributed systems than tracking or reference counting alone.

5.4.3. Weighted reference counts

Using a straightforward implementation of this reference counting scheme two problems remain. Firstly, cyclic structures can only be removed if they do not cross processor

boundaries. Distributed cyclic objects need to be collected with a different method. We will examine this later. Secondly, pure reference counting in a distributed system implies that messages are needed to update reference counts. We need both increment an decrement messages and this may lead to racing conditions. If a decrement message arrives before an increment message, a node might be destroyed that is not garbage at all.

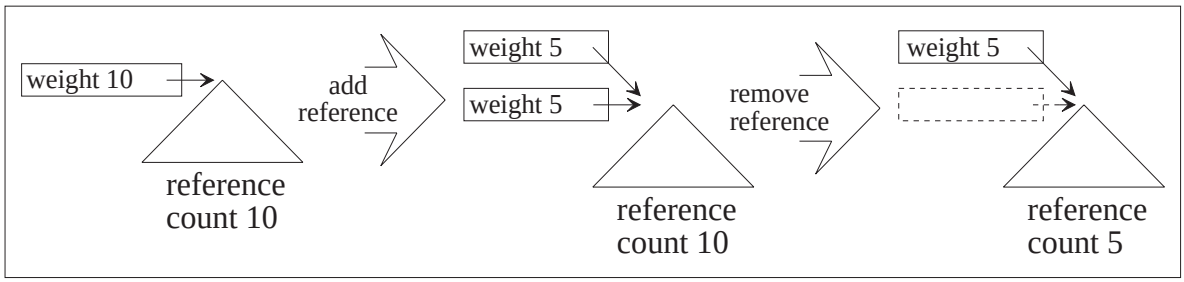


Figure 5-7: *Weighted reference counting.*

This problem can be solved by using a weighted reference counting algorithm (Bevan, 1987; Watson and Watson, 1987). As the name suggests, it assigns a weight to each reference (i.e. each channel node). Initially, the weight will be some huge power of two. The 'reference count' of each node equals the sum of the weights of all references pointing to it. On duplication of a reference, the reference count will not be increased, but the weight will be equally divided over both references. On removing a reference, the reference count will be decreased by the corresponding weight. As always, a node becomes garbage when its reference count drops to zero. As a result, only decrement messages are needed. The main drawback of this method is that indirection nodes are needed when the weight cannot be split any further. In practice this does not happen very often, but in a few cases it might present a problem (see the proposed broadcast mechanism in chapter 8).

Additionally, weighted reference counting introduces a slight practical problem. In addition to duplicating a reference to a remote graph, it is also possible that a new reference is created to a locally available node that already has a global address. If we strive to maintain a single global address for each node we must increase its reference count. This is problematic if such a number is a considerable power of two already. Therefore, we do not preserve unique global addresses, but we create additional ones, each with its own reference count. Consequently, several indirection table entries may refer to the same graph. If multiple requests arrive via various entries, a distinct waiting list element will be stored in the waiting list of the requested graph, each referring to one of these entries (see figure 5.1), so that the updating reducer can find all requests.

5.4.4. Garbage collection as implemented in Clean

Here, we will show how we have extended an ordinary copying garbage collector to realise local garbage collections as described above. It does not remove cyclic structures that are distributed over multiple processors, nor does it detect garbage reducers: only stopped and empty processes are removed. This means that we assume that processes refer to graphs that are connected to the root. If not, they should stop themselves in due time, so that they can be removed. Six phases can be distinguished.

1. The algorithm starts by inspecting the indirection table for live processes. We not only use the indirection table to give global addresses to graphs, but to processes as well. This address is generally referred to as the process id (the pid). An entry may refer to the private heap of a process (see also figure 3-2). From there one can reach the registers and the stacks of a process. The first phase determines which processes should be copied. It removes stopped processes from the indirection table, so that they will be ignored in the next phases. Empty processes are not removed, simply because these are never stored in the indirection table.
2. The second phase copies the indirection table to the new semi-space, possibly after reducing its size.
3. Hereafter, the register sets of live processes are copied. This only applies to off-chip processes. The on-chip registers are not touched at all: they will stay available as they are located outside the heap.
4. Next, the garbage collector copies the stacks and the private heaps (excluding the nodes in it) of all live processes. If necessary, their size will be reduced. All on-chip stopped processes automatically become empty ones, as they are not stored in the indirection table anymore.
5. This phase performs most of the work. For every entry in the indirection table that has a non-zero reference count, it copies the graph it refers to. Additionally it copies the graphs that are reachable from the stacks and the registers of live processes. For this we use a common copying garbage collection mechanism.
6. The last phase examines the original set of channel nodes in the semi-space that has become garbage. If a channel node does not contain a forwarding pointer it has not been copied, so it has become obsolete. In that case, the channel node is transformed into a decrement reference count message and appended to the outgoing message list.

To realise the last phase in an efficient way, we need some structure for tracking all channel nodes that have become garbage in a particular age (the time between two garbage collections). We have achieved this by uniting all channel nodes in a doubly linked list. This list will be preserved until the last garbage collection phase. At that moment it will be traversed to filter out all garbage channel nodes and transform these into decrement reference count messages. Simultaneously a new list will be constructed in the new age.

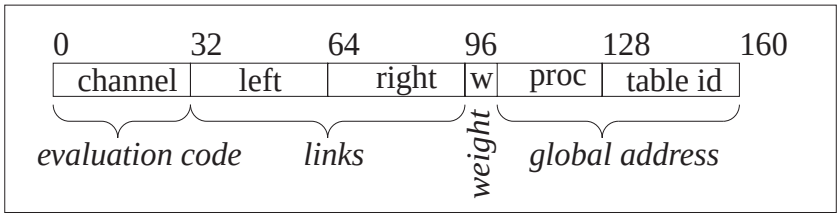


Figure 5-8: *The final layout of a channel node.*

As a result, channel nodes are rather large. The figure 5-8 shows all fields of a single channel node. The two links to the other channel nodes take one word (32 bits) each. To avoid any hard limit on the size of the indirection table - and to enable some experiments with tables of a different structure - one should keep the size of the table id fairly large. We

have reserved an entire word for it. The processor id on the other hand can be slightly smaller, because the number of processors is not very likely to exceed memory size. In addition, the weight *w* can be stored as *log(w)* because it will always be a power of two. Consequently we have reserved merely five bits for the weight and combined this field with the processor id, which left 27 bits for the latter. This still allows for parallel machines with over 130 million processors. Summing up all this, we see that a single channel node takes five words of 32 bits, which is quite a bit larger than the minimum size of function nodes. Consequently, one generally cannot replace function nodes by channel nodes without risking an increase in memory use.

The algorithm above shows the current state of the Concurrent Clean implementation. Clearly it does not remove all garbage, so it is only a partial solution. The removal of distributed cycles and irrelevant processes is still problematic. This is known as a hard problem. Nonetheless, we will have a closer look at this below.

5.4.5. Removing garbage reducers

Garbage processes reduce graphs that are not connected to the root of the computation. They can only be introduced in a system that allows speculative parallelism. In general, the removal of such irrelevant processes is considered to be rather hard. This is one of the reasons that many implementations do not support speculative parallelism at all. However, as we have pointed out earlier (see section 3.2), this form of processing can be very useful. In addition, there are some typical forms of processing in which speculative tasks are able to stop themselves, shortly after they have actually become garbage reducers. Consequently, it will often be possible to exploit speculative parallelism without requiring an additional mechanism to kill irrelevant tasks. They can kill themselves. In spite of this, one cannot always guarantee the absence of garbage reducers if one allows speculation. If we truly want to take advantage of this form of processing, we still need to find a way to remove garbage reducers without introducing substantial overheads.

Clearly, there is a strong relation between the relevance of reducers and the neededness of graphs. On the one hand a reducer is irrelevant if it can only touch nodes that are garbage. On the other hand, live reducers determine the reachability of graphs. This becomes apparent if we consider locked nodes. These do not contain any references to the sub-graphs that are required to compute them. Instead, these graphs are stored in the registers and the stacks of the reducing process (see figure 5-9). In this way they automatically become garbage as soon as the reducer pops them from the stack, and not merely after it updates the locked node. As a result one cannot discard any of the graphs that are reachable by a live process (the nodes that are reachable by process *p* will be referred to as ‘the nodes of *p*’). Consequently, the most promising approach for detecting garbage processes is to link it to ordinary garbage collection.

This is not trivial. Garbage collectors commonly only follow live data. These tracks do not lead to any garbage reducer. This means one would have to find all live processes and discard the remaining ones, similar to ordinary garbage collection. This is problematic as well, because graphs do not refer to reducers in general. No node contains information that reports which process might still reach it. At best, one can only tell which reducer is reducing a particular locked node.

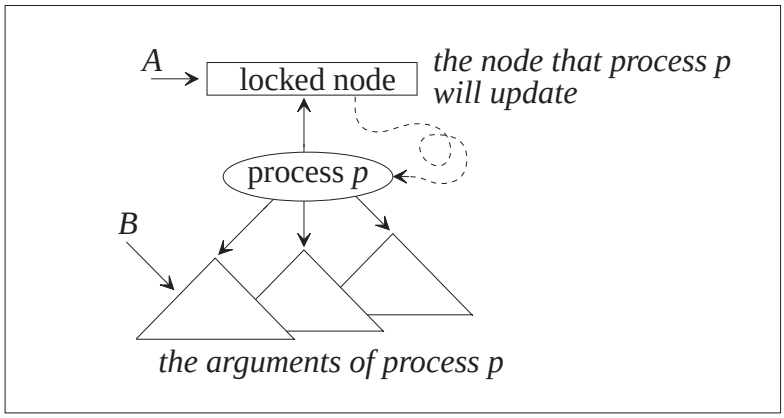


Figure 5-9: *If a locked node is reachable (via A), the process that locked it is reachable as well, and so are its arguments. A garbage collector can easily detect this, provided that it has a way to get from the locked node to the process that locked it (the dotted arrow). In general this is no problem, as a process can leave its mark in each node it locks. However, if merely the argument graphs are reachable (via B), process p may be considered reachable as well, as it might reduce reachable nodes. Unfortunately, the garbage collector cannot quickly detect this: it does not have a link from the arguments to process p.*

If our goal is to find all live processes, it seems we would have to extend ordinary garbage collection in an rather elaborate way. After tracking all nodes that are reachable from the indirection table, one would have to start tracking the nodes of each process. If it turns out that none of these nodes have been encountered earlier, the process might be garbage. However, we cannot be sure until we have tracked all processes. If another live process refers to a node of a process that was presumed garbage earlier, it will not be irrelevant after all and its nodes will have to be preserved. Possibly this will also force rehabilitation of other processes that were accused of being irrelevant earlier (see figure 5-10).

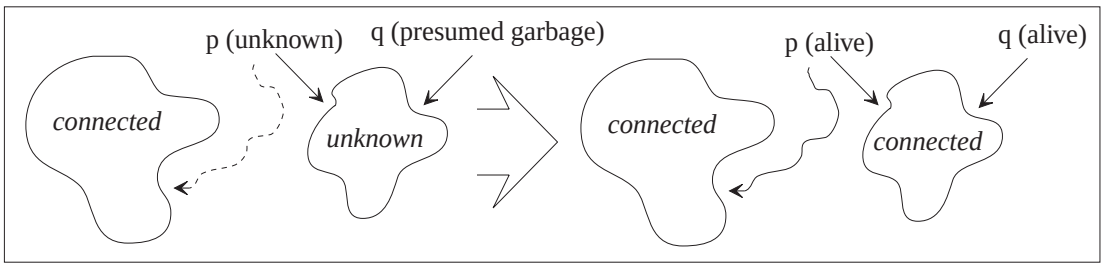


Figure 5-10: *If it turns out that process p is able to reach a connected part of the graph it is not garbage. All nodes that are reachable by p will then become connected as well. This might include some nodes of a process q that was judged irrelevant earlier. If so, q will stay relevant after all.*

This form of garbage collection can be rather costly in some situations. This depends on the order in which processes are considered and the existence of garbage processes. Most of all, it matters whether the root node of a live process is reachable (reference *A* in figure 5-9), or merely some nodes at the leaves (reference *B* in figure 5-9). Usually, the former will be the case. If we start by tracking such processes in the right order, we will quickly detect that they cannot be removed, without temporarily mistaking them for garbage processes. This means we can use an ordinary garbage collection mechanism on these processes, without the need to perform substantial additional checks. In contrast, the classification of the remaining processes will introduce more problems. First of all, we can only discover a - possibly - irrelevant process after checking all its nodes, instead of just one (or a few). Once we have detected a process that does not refer to a connected part of the graph we must take into account that we might re-encounter one of its nodes later, while tracking the remaining processes. To enable detection of such nodes - and of the corresponding process(es) - we possibly have to store extra information in them, requiring additional traversals for restoring them if they turn out to be reachable later.

A much simpler approach is possible if we allow the use of an alternative definition for a garbage reducer. One might consider a reducer to be irrelevant if no node that it has *locked* is reachable from the root of the computation. Removing such a reducer is safe, as the final result does not depend on updating any of these locked nodes. Any process that gets blocked on a garbage locked node must be garbage as well.

Note that we will remove more processes than before. A reducer that has not locked any reachable nodes will be killed, even though it might still evaluate a reachable one later. This is not as odd as it might seem: if a reducer is started on some graph, the first thing it does, is locking the root node (consider also figure 5-9). This node will eventually be updated with the result of the computation, directly followed by the death of the reducer. If at some point before its update the root becomes garbage, the speculative computation as it was meant to occur will not be of any interest anymore, even though some sub-computation might still be useful. The new definition avoids keeping a large computation (and the associated data) merely because a small part of it is still needed. If it the sub-computation remains needed it will automatically be computed by a separate process later.

Note that similar arguments hold for other forms of processing. A stream process, for example, will not die after evaluating a node, but continue with the evaluation of other nodes. Such a process will have locked its continuation node by the time it updates the root. The continuation node will then become the new root, which has to stay reachable in order for the process to be useful.

This has important practical advantages, because detecting garbage processes becomes much easier. Note that it is possible to adapt locking of a node in such a way that it becomes possible to identify the reducer that locked it (the dotted arrow in figure 5-9). This does not have to introduce any additional costs. If we use marks to distinguish live processes from irrelevant ones we can use the following (local) garbage collection algorithm.

- First, unmark all reducers. For a start, they will all be considered garbage.

- Next, copy all nodes that are reachable from the indirection table (i.e. perform phase 5 of the garbage collection algorithm presented above). Each time a locked node is encountered, the reducer that locked it will be marked.
- Now, repeatedly pick out a marked reducer and copy its nodes. Continue marking the corresponding reducers if locked nodes are encountered. Repeat this step for as long as there exist marked reducers whose nodes have not been tracked yet.
- Finally, remove all unmarked reducers. These are all garbage according to the new definition, because the earlier steps have uncovered all reachable nodes, and thus all reachable locked nodes. If a process had any reachable locked nodes it will have been marked.

This algorithm exactly traces live processes in the 'right' order, just as the original one might have done in its efficient first phase. The rest of the processes is garbage by definition. So, we avoid the costly second phase that checks all nodes of a presumed garbage process.

In conjunction with fair scheduling this form of garbage collection avoids many of the problems that are pointed out by Peyton Jones (1989-c). First of all, it is not possible that a speculative task is removed when it is reducing a node that is reachable by a conservative one. The locked node will not be garbage in that case, and neither will be the speculative task. Secondly, there are no problems related to priority upgrades, simply because we do not need any (see section 3.2.2). And finally, we do not consider the problems that are introduced by space leaks a legitimate argument against speculative parallelism. Space leaks need to be avoided in the first place.

Still, one problem remains. As we have indicated earlier, there may be some delay between detecting and removing garbage. An irrelevant process might live on in its children, by quickly spawning new processes on other processors. Each new process does not even have to lock a node - these are garbage anyway -, but it merely has to start up a new process at another processor before the trail of local garbage collections behind it catches up. Doing so, it might stay ahead of the garbage collector indefinitely. One may question whether such processes will be started up speculatively. Especially if the dangers are so obvious it seems rather awkward to start up quickly spreading processes in a speculative way. Nonetheless, we will have a closer look at this in the next subsection.

5.4.6. Removing distributed cycles

Cycles that are spread over multiple processors cannot be reclaimed by reference counting alone. Still, reference counting is very suited for collecting non-cyclic structures. Therefore it is desirable not to discard reference counting, but to introduce a complementing mechanism which sole purpose is to remove cyclic structures.

This supplementing garbage collector probably will not have to remove distributed cycles extremely quickly. In many cases, distributed cycles will be rather exceptional, so that the original garbage collection methods will quickly free enough memory to maintain ordinary operation in the short term, while the gradual removal of distributed cycles will avoid space-leaks on the long run. The effectiveness of such co-operation however, depends on the characteristics of the application, so it remains to be seen whether a relatively slow solution for removing distributed cycles suffices in general.

If so, it will be worthwhile to reconsider distributed mark-scan algorithms. More precisely, we will mainly need to focus on the marking phase. Nodes left unmarked may simply be given a reference count of zero, so that subsequent local garbage collections will remove them. To avoid stopping all processors, a distributed marking algorithm will have to run concurrently with normal computations. This can be achieved for example, by using the local compacting garbage collections for advancing the marks (in the same way as it is used for introducing decrement reference count messages). Note that it is possible that the reference counting mechanism removes some nodes - either marked or unmarked - during a single marking phase. This may reduce the number of nodes that yet have to be visited by the marking algorithm.

The most evident question in such a system is when to stop marking nodes. Two alternatives come to mind. For one, it is possible to proceed marking for as long as there exist nodes that have not yet been visited. However, ordinary reducers are continuously introducing new nodes and it is most likely that the marking phase cannot keep up, unless all processes get suspended due to shortage of memory. If few cycles exist, no memory problems will occur, but the marking phase will not complete and cycles will prevail. If many cycles exist, memory shortages will arise, leading to a trashing behaviour and possibly to a full stop of the system. The marking phase will then be able to catch up, but this will not be much better than stopping the whole system in advance.

Alternatively, the marking phase may stop at nodes that have been introduced since it started. Only old nodes will be visited. New ones will not be considered garbage, nor will the old ones that are reachable by new nodes. If no speculative parallelism exists, this premise is rather harmless. We will assume this for the moment. Now, we can simply presume that all nodes that are reachable by some process are also connected to the root. New nodes can only be created by processes, so they will not be garbage at the moment they come into existence. Consequently, marking not only may start at the root, but at each process as well, thus accelerating its completion. New nodes will be created with a mark in them. This marking algorithm will surely stop, but clearly, not all garbage cycles will be detected in a single run of the marking phase. Nonetheless, if a garbage cycle remains undetected in the current marking round, it will be reclaimed in the next.

The existence of speculative parallelism complicates matters. We do not like to start marking nodes from irrelevant processes. In some cases we can prevent this. If we consider the algorithm for removing garbage reducers that has been presented above, we see that we do not have to start marking nodes from garbage processes that can be spotted by this algorithm. There are some cases however, that this method cannot detect. First of all, the original difficulty of detecting garbage reducers remains. An irrelevant process might simply outrun the garbage collector by quickly starting up new processes. Secondly, if a garbage reducer refers to a distributed garbage cycle (i.e. if it is ‘located’ on a cycle) it will not be recognised as being irrelevant. Both the cycle and the process will persist, provided that the process does not stop itself and that at least one node on this cycle - or in another reachable part of the graph - has been locked by the garbage process. Otherwise, the reducer will be considered garbage even though it effected marking of the cycle. The cycle itself can then be reclaimed later, if it is not reachable by any other process. Note also that, if locking is the result of the evaluation of a node on a cycle, the problem might disappear

automatically. Reductions on a distributed cycle might move the cycle to a single processor, which makes it an easy prey for local garbage collections.

An idea for solving these problems is to limit the life span of processes and to extend the garbage collector with life-giving capabilities. Processes will only be allowed to run for a limited period of time, that is, for as long as they do not run out of 'fuel' (Haynes and Friedman, 1984; Wong and Yuen, 1992). To ensure that processes cannot live on indefinitely by means of their children, they have to divide their energy over their off-spring. This does not have to be a strict division: it is sufficient that children are started with less energy than their parent. Eventually, new processes will not get any initial energy, so they will suspend immediately without any chance of spawning any children. In this way the problem of unbounded spreading of processes is avoided. The garbage collector on the other hand, will refuel any process as soon as it is able to verify that it is not garbage. Additionally, it will not start marking from processes that have run out of fuel. These have a relatively high probability of being garbage, and at the same time they cannot create any new nodes, so if they are not encountered in a single marking phase they can be reclaimed.

Unfortunately, it is not so easy determine which processes should be refuelled. At any moment in time some garbage process might still have some fuel left. Marking will then spread from such a process and thus, we cannot simply refuel all processes we encounter during marking. The only processes that are certainly not garbage, are those that are directly reachable from the root of the computation. However, if we only refuel these processes, we loose parallelism. So, we need a solution that falls between both extremes.

One possibility to realise this, is to allow processes to pass energy to processes on which they depend, with the restriction that the total amount of energy does not increase. Again, we might use the marking mechanism for passing energy from one process to another (but of course, processes can also do this themselves). If a process passes some amount of fuel to another one, its own fuel level must drop by the same amount. As a result, a certain branch of computation - possibly consisting of many processes - will certainly die if at some point in time all connections to the root are cut off. The total amount of energy in such an isolated subsystem will gradually drop to zero. Usually, parents will fuel the children they started, but the converse is also possible if a child depends on some shared argument that is - partially - computed by the parent.

Note that we cannot simply give new processes a slightly lower fuel level than its parent. In contrast with the earlier situation, the energy of new processes must be drawn from the energy of the parent. Otherwise, a process with energy level n might start up another one with energy level $n-1$. Subsequently, the parent is able to pass one energy unit - or more -, and then we have got another process with at least energy level n. This can repeat itself indefinitely.

These techniques have not yet been implemented, mainly because they are quite complex and it would take too much time to realise them. This means that we do not know whether they are able to remove garbage with sufficient speed. Furthermore, it is unclear in which ways ordinary reductions are influenced by the fuelling system introduced above. However, our point is that there is no reason to assume that distributed cycles and speculative parallelism are too difficult to handle. This still remains to be verified.

5.5. Performance measurements for parallel programs

Although our garbage collector does not remove distributed cycles and irrelevant speculative processes this usually does not impede the execution of test programs. Weighted reference counting is sufficient in many cases. Table 5-1 lists the programs we have tested initially. These indicate which algorithms give significant parallel speed-ups, and which constitute a problem. This section will shortly consider the source of the problems, and it will compare our results with those of other parallel implementations of functional languages.

Table 5-1: *Execution times of some well-known parallel test programs in Concurrent Clean. The nfib benchmark computes nfib 30. Sieve computes 10,000 primes. Queens computes the number of solutions of the queens problem on a board of 10 by 10. Rnfib is similar to nfib, only now floating point numbers are used instead of integers. The fast fourier programs has been executed on a list of 8000 complex numbers. And finally, mandelbrot computes a well known fractal in a resolution of 560 by 320 pixels and depth 128.*

program	number of processors					
	1	2	4	8	16	32
nfib 30	12.2 sec.	6.5 sec.	3.5 sec.	2.2 sec.	1.4 sec.	1.1 sec.
sieve	19.4 sec.	31.2 sec.	32.1 sec.	23.6 sec.	16.9 sec.	14.5 sec.
queens	47.9 sec.	28.5 sec.	15.1 sec.	9.0 sec.	6.2 sec.	4.7 sec.
rnfib 30	23.7 sec.	12.2 sec.	7.1 sec.	3.9 sec.	2.2 sec.	1.6 sec.
fast fourier	13.8 sec.	11.2 sec.	8.7 sec.	6.3 sec.	5.6 sec.	5.6 sec.
mandelbrot	147.0 sec.	91.0 sec.	54.3 sec.	34.3 sec.	18.2 sec.	10.6 sec.

As we can see in table 5-1, some programs give rather promising speed-ups, while others hardly give any gain in performance at all. The sieve and the fast fourier program contain less inherent parallelism than ordinary divide-and-conquer programs like the other benchmarks. In addition, both programs use asynchronous *pipelining* (the fast fourier program uses a pipelined merge function). The problems are mainly related to overheads in communication. We will take a close look at these efficiency problems in the chapter 7 and 8.

The programs that do perform relatively well exploit divide-and-conquer parallelism (nfib, rnfib, queens, and mandelbrot). For these programs we used thresholds to control the grain size of computation. However, speed-ups are not linear. This is caused by load imbalance. The processor that gets most work determines the overall execution time. This situation becomes worse as runtimes get shorter: a small deviation in load (in absolute terms) will then have a significant effect on performance.

We have not (yet) employed any load balancing technique for the divide-and-conquer programs. Instead, processes have been scattered randomly over the network. This somewhat balances the load if the number of processes is high enough. Therefore we had to

set the threshold such that enough processes were introduced to balance the load, while also avoiding an abundance of processes, which would introduce too much overheads.

Unfortunately, mainly these fairly simple divide-and-conquer programs get most attention in reports on parallel implementations of functional languages. This is even the case if one uses an interpreter, for which communication overheads are rather irrelevant. Note also that, if one uses code generation, one commonly does not have much trouble with these divide-and-conquer programs either. The speed-ups can be kept comparable to interpreters if one scales the problem appropriately, so that computation and communication do not digress drastically.

If we consider the results that have been presented for the HDG machine (Kingdon, Lester and Burn, 1991) we only encounter nfib, the takeuchi function, and the queens program (see table 5.2 for the nfib figures). The listed execution times are very short, as the problem sizes have been kept quite small. More complex programs cannot be tested on the HDG implementation, because it lacks a garbage collector. As we have seen in chapter 4, the HDG machine produces less efficient code than the Clean compiler. Consequently, having very short sequential execution times, the Clean compiler has difficulties to obtain similar speed-ups for these tiny programs, although the absolute execution times of the parallel Clean programs are always smaller than the HDG programs. For larger problem sizes the speed-ups in Clean become comparable to those of the HDG machine.

Table 5-2: *Parallel performance of the HDG machine and Clean for the nfib benchmark*

	1 processor	2 processors
HDG	1.28 sec.	0.68 sec.
Clean	0.12 sec.	0.08 sec.

An interesting point about the HDG machine is that it automatically introduces parallelism. No annotations, nor any thresholds are needed. In addition, the HDG machine employs some heuristics to balance the load. Unfortunately, the effectiveness of these techniques is only demonstrated for very small examples, for which the use of annotations is no problem at all. And, as Concurrent Clean demonstrates, the use of a random process allocation does not give significantly worse execution times, compared to using the HDG load balancing heuristic.

The parallel speed-ups of the partial implementation of Concurrent Clean on the ZAPP architecture are fairly similar to the figures of table 5-1. Table 5-3 lists the parallel results on ZAPP for nfib, queens and matrix multiplication. The Clean figures for matrix multiplication can be found in chapter 7 and 8. See also table 4-7 for a comparison of absolute performance figures between our implementation and the ZAPP implementation. ZAPP does have slightly better speed-ups for nfib. This can largely be attributed to the special support for divide-and-conquer parallelism. ZAPP does not support other forms of parallelism, such as stream processing.

Table 5-3: *Parallel performance of the ZAPP implementation.*

	1 processor	8 processors
nfib 30	12.1 sec.	1.69 sec.
queens (on a board of 8×8)	2.36 sec.	0.42 sec.
matrix multiply 64×64	5.79 sec.	1.67 sec.

And finally, as we have seen before, SkelML lists some results for a ray tracing program. For SkelML the same applies as for the HDG machine. It is slower than Clean, but it has similar speed-ups if the problem size is scaled appropriately.

Table 5-4: *Parallel performance of SkelML and Clean for a raytrace benchmark.*

	1 processor	16 processors
SkelML	8.70 sec.	0.83 sec.
Clean	2.31 sec.	0.60 sec.

Concluding, the Clean implementation performs well compared to other transputer implementations. The limitations of the garbage collector have not yet prevented execution of test programs. However, some algorithms do not give significant speed-ups. These programs are often ignored by other implementations, partly because they only support divide-and-conquer parallelism (or more precisely, because they do not support pipelining), and partly because of other limitations, such as the absence of a distributed garbage collector.

6. The influence of Graph Copying on Runtime Semantics and on Uniqueness Typing

---

In the previous chapter, we have explained how to copy graphs. We have seen that a redex can have a defer attribute, and that copying stops at such a node. These are rather technical matters. We have not yet considered a more fundamental issue. Which redexes should be deferred? By answering this question we devise a *copying strategy*, which tells the graph copier which redexes to copy, and which not.

Not all copying strategies are equal. First of all, they influence the runtime semantics. Depending on the copying strategy (a potentially infinite amount of) work will be copied, moved, or evaluated locally. Clearly, this can greatly influence performance. Therefore, a copying strategy should be chosen such that programs maintain a clear runtime behaviour. Secondly, as we will see in this chapter, some copying strategies are incompatible with uniqueness typing. This is possible, because uniqueness typing is defined for the *standard* graph rewriting semantics, while graph copying forms an extension of these standard semantics.

This chapter has been structured as follows. Section 6.1 will give a short overview of uniqueness typing. Section 6.2 will identify the conflicts between uniqueness typing and graph copying. In particular, it will show that the copying strategy that has been employed in Concurrent Clean so far is incompatible with uniqueness typing. Section 6.3 will identify possible solutions. Basically, these can be divided into a group that does not alter the copying strategy, and a group that does. Section 6.4 will present the solution we have adopted: a different graph copying strategy called *lazy normal form copying*. This strategy both avoids uniqueness conflicts *and* it improves the runtime semantics of programs. In section 6.5 we will have a close look at one aspect of this strategy, namely the copying of work. Section 6.6 will show what the new copying strategy implies for the semantics of some example programs. And finally, in section 6.7 we will list our main conclusions.

6.1. Uniqueness typing

Uniqueness typing (Barendsen and Smetsers, 1993, 1995-a and 1995-b) is an important techniques to efficiently implement functional languages. The uniqueness type system is an extension of a classical type system. It uses knowledge about the standard semantics of

graph rewriting and the reduction strategy to derive information about sharing. This makes it sometimes possible to update objects destructively without consequences for referential transparency (compile-time garbage collection). Thus, it can optimise standard graph rewriting, which forms the basis of efficient array implementations and the Concurrent Clean I/O system. In this section we will give a short introduction of uniqueness typing.

The Concurrent Clean uniqueness type system is based on the observation that a node *a* can be *unique* with respect to a function node *f*. This is the case if *a* can only be reached via *f* and there exists only one path from *f* to *a*, as depicted below. As we can see, uniqueness is a local property: the function itself may be referred to by many others, and many paths may lead from the top of the graph *G* to *a*.

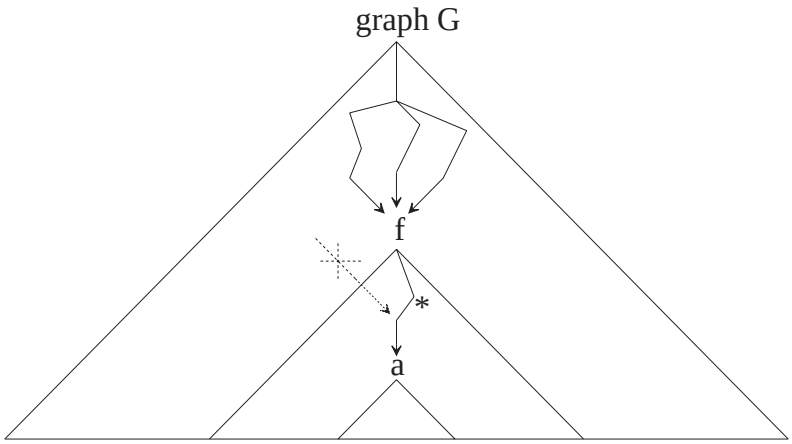


Figure 6-1: *An argument a is unique with respect to f. The star indicates a unique path.*

6.1.1. Functions and uniqueness propagation

Despite the locality of the uniqueness property, it can be used to ensure that *f* has exclusive access to *a*, because no other function is able to reach *a* it without evaluating *f* first: one cannot traverse a function node during normal reduction, simply because one cannot match on functions, but only on function results (i.e. constructors). So, functions shield their unique arguments from the rest of the world.

Functions do not only obstruct the upward propagation of uniqueness, they also block downward propagation. A function with ordinary non-unique arguments can deliver a unique result. This is rather trivial. Using the information of non-unique arguments a function can create a totally new object. Clearly, such an object is unique to the function itself, and it can be delivered as a unique result. In brief, functions obstruct the propagation of uniqueness information. (see figure 6-2).

6.1.2. Constructors and uniqueness propagation

For constructors different rules apply. Unlike function nodes, constructors do not shield their arguments from the rest of the world. One can match on a constructor and thus reach its arguments from ‘outside’. Consequently, it is not very sensible to consider uniqueness of arguments with respect constructors. An argument of a constructor can only be unique if

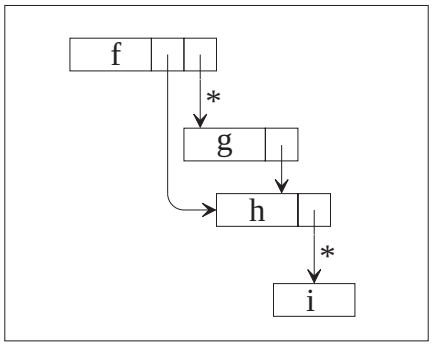


Figure 6-2: *Function nodes obstruct propagation of uniqueness information downwards, as well as upwards. The star indicates uniqueness. Assume g builds a unique result, but it does not have a unique argument. In contrast, h has a unique argument, but it does not deliver a unique result. The expression f c (g c) with c defined as (h i) results in a graph as depicted. The graph rooted by g is unique, but the sub-graph rooted by h is not. In short, uniqueness of a graph does not imply that sub-graphs are unique, and uniqueness of a sub-graph does not mean that surrounding nodes are unique.*

the surrounding constructor is unique. In figure 6-1, the nodes between *f* and *a* must all be unique with respect to *f*. Clearly, *a* could not have been unique with respect to *f* otherwise. The converse *is* possible: the argument of a unique constructor does not have to be unique. For instance, the nodes that are reachable from *a* in the picture 6-1 do not have to be unique with respect to *f*. Briefly, constructors propagate the uniqueness property upwards, but not downwards.

As we will see later, virtually the same rules apply to curried functions On a low level, curried functions are actually structured objects. Consequently the rules for curried functions are very similar to those for constructors. However, to avoid confusion, we will only consider constructors for the moment.

6.1.3. The uniqueness type system

The main consequence of the propagation rules above, is that any uniqueness information about some graph (with respect to an enclosing function), can only reveal something about the final result of the graph. Function nodes block propagation of uniqueness information, so at best, uniqueness properties give some information about the topmost function nodes in a graph, but not for function nodes contained within others. Therefore, the uniqueness properties of a graph are closely related to its type.

In general, uniqueness of function arguments cannot be decided at compile time. The Concurrent Clean system therefore incorporates a decidable approximation that uses *unique type attributes*. By default, the type system infers these attributes, but one can also attribute function types explicitly. The unique type attribute indicates that an object is not shared. Initially, each object gets this attribute when it is created: only the creating function has a single reference to this object then. However, as soon as an object becomes shared (for instance by referring to it twice in some expression) it looses its unique type attribute forever. The formal arguments of a function may have a unique type attribute as well (this

can be indicated by the programmer). Whenever a formal function argument has the unique attribute the type system guarantees that the function has private access to the argument: it does not allow applications of the function to non-unique objects.

There can be many references to a unique argument after - and even before - a function accesses it, but the uniqueness type system ensures that uniquely attributed arguments have reference count one at the moment the function inspects them (amongst others it takes into account the order of evaluation). This allows compile-time garbage collection for unique arguments. If a unique argument is not part of the function result it becomes garbage, and consequently the compiler can reuse the freed space for constructing the function result.

6.1.4. Type coercion

In contrast to pure 'linear' systems, unique objects can be coerced to non-unique ones (Note that the converse is not possible). This happens at the moment a function introduces several references to the same unique argument. For example, it may simply pass the same argument to two functions. This will have no effect on the outcome of the program. The only difference is that certain optimisations are not possible on non-unique objects.

Some unique objects cannot be coerced to non-unique ones. They are *essentially unique*. Any attempt to share such an object must be rejected by the compiler. We will see some examples of essentially unique data later in this chapter. It will become clear that copying conflicts are especially hard to avoid for these objects. On the other hand, it will also become apparent that essentially unique data cannot be avoided in certain situations (see section 6.5.1, which explains currying in Concurrent Clean).

6.2. The conflict between lazy graph copying and uniqueness typing

Lazy graph copying is an extension of standard graph rewriting semantics. Therefore, the question arises whether it is compatible with the optimisations that are introduced by the uniqueness type system, which only takes into account standard graph rewriting semantics. At first sight, if one considers the meaning of the terms 'uniqueness' and 'copying', maintaining consistency seems problematic. How can some unique data structure remain unique after it has been copied? If this cannot be guaranteed, one cannot take advantage of uniqueness properties in a parallel environment, in particular one cannot apply important optimisations. This would seriously degrade performance, and the possibility to efficiently incorporate arrays and I/O facilities.

In this section we will identify in which ways the graph copying extension invalidates derived uniqueness information. Clearly, not all extensions are dangerous. If copying changes a non-unique object into a unique one (at runtime), there is no problem, except perhaps for not exploiting this information (this might be an interesting optimisation, but we will not consider this here). In contrast, if graph copying changes a unique object into a non-unique one, this will result in serious conflicts if this change is not taken into account by functions that refer to this object. Objects that are actually not unique might then be updated in place, which destroys referential transparency.

6.2.1. The source of the conflict: deferred unique objects

Interestingly, eager graph copying does not impose any problems. If one copies a graph as a whole, the copy will be exactly the same as the original. For both the copy and the original the (local) uniqueness properties of individual nodes will be the same. If an argument is unique with respect to some function, the copy of the argument will be unique with respect to the copy of the function.

Problems arise when introducing laziness to copying. Lazy copying will not always result in a physical copy. Instead, it will introduce new references to deferred nodes, and if these have a unique type this may invalidate the uniqueness information that has been derived by compiler. So, deferred nodes can be problematic if they have a uniqueness attribute. This means that conflicts are related to the copying strategy, which determines which nodes should be deferred and which not.

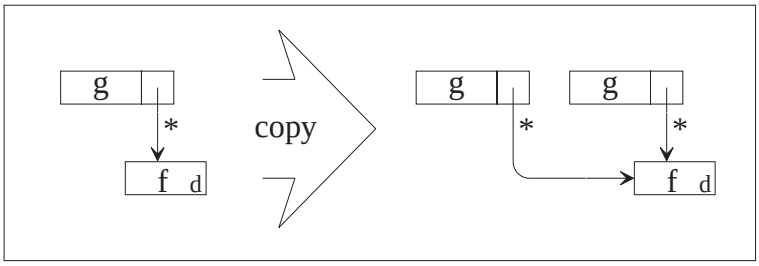


Figure 6-3: *Lazy graph copying affects actual uniqueness properties, without changing the derived uniqueness information. Suppose that f is a function that delivers a unique result and g takes this unique result for its argument. If we have the following expression (h,* ***{P}*** *h), with h defined as g* ***{I}*** *f, the function node of f will be deferred - as it will be reduced by a separate process - so copying will stop at f. The leftmost figure shows the graph h before copying, the rightmost figure shows the original h and its copy. The star indicates derived uniqueness (which does not change during copying), but clearly f is no longer unique with respect to g after copying.*

The question is, whether such a conflicting situation can actually occur during the standard lazy graph copying strategy of Concurrent Clean. Unfortunately this is the case. Standard lazy graph copying will defer locked nodes, and nodes on which a process has been started. Thus, it avoids copying of nodes that would otherwise certainly introduce duplication of work. Both types of deferred nodes can have a unique type attribute, and, as we can see in figure 6-3, annotated nodes can introduce a conflict. Interestingly, only annotated nodes are problematic. If locked nodes are unique no conflicts arise. We will explain this is section 6.2.3.

In this thesis, we will refer to the standard graph copying strategy of Concurrent Clean as (plain) lazy graph copying, although this term generally does not indicate a copying strategy, but the class of copying algorithms that stop copying at certain nodes. However, no generally accepted terminology has been established yet for copying strategies.

6.2.2. Traversing function nodes during copying can be dangerous

Not all deferred unique nodes will introduce conflicts, but only those contained within a function node that gets copied. If no function node around a deferred unique graph gets copied (see figure 6-4) no problems occur due to the upward uniqueness propagation of constructors: all copied nodes around the unique part will be unique as well. This means that the original graph will become garbage after copying has succeeded, because the copy function delivers the copy and discards the original. Therefore the newly created pointer to the unique deferred substructure will become the only one left after copying and uniqueness is not violated. This shows that conflicts are closely related to the exclusive ability of the graph copier to traverse function nodes.

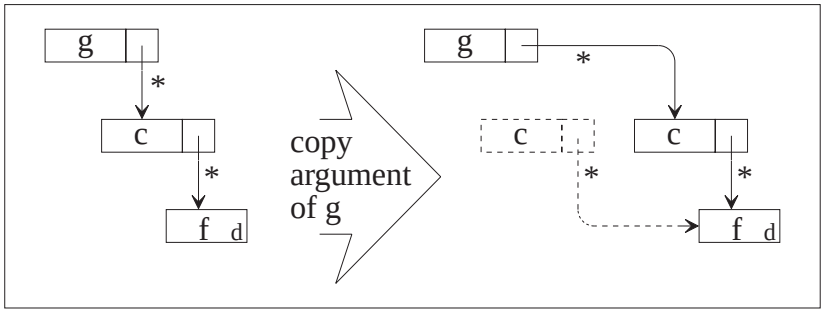


Figure 6-4: *Suppose that c is a constructor. If f is unique, c has to be unique as well. Consequently, if c gets copied, the deferred unique node f will not cause problems, because the original argument of g becomes garbage.*

As long as no function nodes are copied no conflicts occur, but if one does not copy function nodes one cannot start functions at another processor. Not all function nodes are dangerous however. First of all, only by traversing a function node with unique arguments one can add references to its unique arguments (there is no other way to reach the arguments). And secondly, if one can assure that the original function node becomes garbage after copying, the total amount of pointers to any unique deferred argument will not change. This situation is equivalent to that of figure 6-4, if one replaces *c* by a function node.

In short, deferred nodes that are unique with respect to some function *f* can only give rise to copying problems if the enclosing function *f* is copied *and* if *f* does not become garbage after copying. These observations will turn out to be important for the solution we have adopted.

6.2.3. Locked unique nodes are safe

The observations above imply that locked unique nodes are safe with respect to copying. To explain this, we need to take a closer look at the order in which nodes get evaluated. When a reducer reduces (and locks) a unique argument during evaluation of some function *f* it will have locked the surrounding function node *f* as well. Any access to the unique argument must pass *f*. This holds for the graph copier as well, and as the parent function node is locked, copying will stop there automatically and not at the argument. So, the normal functional reduction strategy will never allow the graph copier to traverse nodes

with locked unique arguments. Consequently, it is safe to defer nodes during ordinary sequential reduction. Clearly the order of evaluation is crucial here, considering the problems that occur when annotations for parallelism are used.

6.2.4. Introducing additional deferred nodes at runtime

The standard lazy copying strategy only defers locked nodes and nodes with an annotation for parallelism. But sometimes, it is very useful to defer additional nodes at runtime. For example, it may be useful to stop copying a graph if it is about to become very large (see also section 7.4). This may keep down communication overheads (if certain parts of the graph are not needed), while also avoiding potential memory management problems. Likewise one may like to stop copying when a large flat data structure - such as a strict array - is hit. Such a data structure will then be transmitted later in a separate message, which can be more efficient than packing it in a message along with the other nodes of the copied graph (we will see an example in chapter 7). And finally, as we will see in chapter 8, it may be useful to keep certain parts of a data structure at a particular processor. To realise the intended behaviour one needs to defer certain arguments at runtime, so that they do not get copied to the 'wrong' processor.

All these 'optimisations' concern data structures. But, chapter 1 has already made clear that the defer attribute has originally been introduced to limit the copying of work. Therefore, only function nodes can be deferred. Still, this is not a very fundamental limitation. In Clean, one can easily defer copying of data (at runtime) by inserting deferred indirection nodes in some data structure. This is correct, because no function will ever assume that a certain amount of data will get copied. The standard evaluation mechanism always takes into account possible indirections (such as channel nodes).

The point is, that there will be several situations in which one would like to insert additional deferred nodes, depending on runtime conditions. Potentially, they all may lead to conflicts with respect to uniqueness. If we devise a solution, we should consider its tolerance with respect to inserting additional deferred (indirection) nodes.

6.3. Potential solutions

Excluding uniqueness typing is not a realistic option to avoid conflicts between uniqueness typing and graph copying, nor is banning lazy graph copying on machine with distributed memory. Keeping both, the current copy strategy is able to change the actual uniqueness properties of graphs at runtime if deferred unique objects are encountered. Three possible remedies come to mind.

- First of all one could implement runtime coercions to make functions aware of the changes that have occurred.
- Secondly, one could avoid the creation of deferred unique objects, that is, objects that may introduce uniqueness conflicts.
- And finally, one could change the copying strategy so that it cannot change uniqueness properties at runtime.

6.3.1. Runtime coercions

The first solution would imply that functions invoke evaluation code that depends on the actual runtime uniqueness properties. The code addresses of function nodes (see chapter 3) would have to be modified by the graph copier if it detects that an argument is no longer unique. Not only is this needed for the nodes in the copy but for nodes in the original graph as well. Such a solution is intolerable. Not only does it make copying considerably more complex, requiring expensive runtime operations for deducing uniqueness properties and adjusting nodes, but it also gives rise to unclear runtime behaviour. This surfaces most clearly when essentially unique objects are involved, or functions that do not have an equivalent with non-unique arguments (for example, functions that write to a file). These cannot be coerced to a non-unique type and a runtime error would be the result.

6.3.2. Avoiding deferred unique objects

Only deferred unique nodes may cause copying problems. Avoiding creation of these objects will solve the copying problems. However, in some situations it is worthwhile to have deferred unique objects and there is no reason that they will actually result in conflicts (i.e. if they do not get copied), so avoiding them regardless will often be harmful and of no use at all. Keeping this in mind, we will examine the feasibility of this method in more detail below.

It is technically possible to avoid creation of unique deferred objects (that is, without throwing away uniqueness typing altogether). The standard copying strategy only defers locked nodes and annotated ones. As we pointed out above, locked nodes are no problem, and the compiler can easily reject the use of unique types for annotated nodes. Unfortunately, such a strategy has three serious disadvantages.

First of all, one risks a notable performance penalty if processes are not allowed to deliver unique results. The next chapter will show that arrays can be crucial for good parallel performance. But, to implement arrays efficiently, uniqueness typing is invaluable. If we do not allow processes to deliver unique results, they would not be able to deliver arrays that can be updated in place. In addition, it would become virtually impossible to implement an efficient distributed I/O system. For instance, processes would not be able to deliver (remote) files that can be modified in place.

The second disadvantage of this solution is that it does not allow additional deferring of unique nodes at runtime (see section 6.2.4.). Apart from the potential loss of performance (and flexibility) that is caused by the absence of these unique deferred objects, this solution requires a runtime mechanism that checks the actual uniqueness properties of nodes. This may be rather hard to realise. Furthermore, one faces the problem of unclear runtime copying semantics, if some nodes get deferred, while others do not, depending on actual uniqueness properties.

The last observation leads to the third disadvantage of avoiding unique deferred objects: rejecting uniqueness for (annotated) deferred objects at compile-time can also be confusing for the programmer. The ***{I}*** annotation in itself has no influence whatsoever on the uniqueness properties of graphs: no two processes running interleaved at the same processor are able to access unique arguments of the same function node, because function nodes are locked during reduction. This is in accordance with the view that processes do

not introduce new data dependencies. They merely provide safe eager evaluation (safe with respect to program termination). In effect, the introduction of processes can - and should - be seen completely separate from the (unwanted) effects of graph copying.

From the above, it will be clear that avoiding the creation of unique deferred objects has serious drawbacks, mainly because these objects are very useful to have for various reasons, while they may not become problematic at all (i.e. if they do not get copied).

6.3.3. Changing the copying strategy

Until now, we have not paid much attention to different copying strategies. There are basically three issues that are important in this respect. First of all, some copying strategies are more efficient (and flexible) than others. We have already seen some examples in section 6.2.4. In this respect, it is important that a copying strategy allows the introduction of additional deferred nodes at runtime. Secondly, different strategies imply different runtime behaviour. Depending on the copying strategy work will be moved, copied or evaluated locally. A good copying strategy should have clear runtime semantics. And finally, as we will see below, some copying strategies do not conflict with uniqueness typing. In the next section, we will present such a safe copying strategy: *lazy normal form copying.* The last part of this chapter (sections 6.5 and forth) will show that it has clear runtime semantics as well, while the remaining part of this thesis will show that considerable parallel speed-ups can be obtained with this new copying strategy.

6.4. A safe copying strategy

In this section, we will present a graph copying strategy that does not alter uniqueness information. Laziness will not be discarded. In contrast, copying will become even lazier as it will stop *before* it hits nodes that may introduce conflicts. Our solution is based on the observations of section 6.2.2: deferred unique objects can only become problematic if the enclosing function is copied, *and* if this function does not become garbage after copying. In other words, a function node can be copied safely if:

1. The function node does not have any unique arguments.
2. The function node does have unique arguments, but one can assure that they will be copied completely, i.e. they do not contain any deferred nodes.
3. The function node does have unique arguments with deferred nodes, but the function node becomes garbage after copying.

The first property can easily be checked by a copying algorithm if the compiler includes uniqueness information in function nodes. The other properties impose more problems. In general, the compiler cannot derive this information. It does not know if a graph can be copied entirely. Amongst others, this would involve knowing whether parts of the graph are being reduced and at which processor they are. In addition, the compiler cannot determine for a particular graph which of its function nodes will become garbage after copying. Most likely it does not even know which function nodes it contains and where they are located, and even if it did, the uniqueness type system only provides

information for the topmost function nodes and not for function nodes contained within others.

It may be possible to derive such information at runtime, in conjunction with complex compile-time analysis, but this will make reasoning about copying behaviour an impossible task. This is especially problematic for function nodes as these represent work. Unclear copying semantics will make it hard for a programmer to figure out what will happen where. Therefore, we have adopted a much simpler solution. This will be presented below.

6.4.1. The lazy normal form copying strategy

We have taken a radical approach that does not copy *any* function nodes (work) at all, unless explicitly requested. By default, copying will stop at every function node. If a programmer wishes to copy function nodes she has to indicate them explicitly. This could be done with annotations, by means of a special type denotation, or otherwise. We propose a new ***{P^n}*** annotation that will copy only *one* function node: the annotated function node at the root, provided that the compiler is able to verify compatibility with uniqueness typing (this will be explained in the following subsection). Copying may proceed at the arguments of a root node, but only as long as they are in root normal form.

Take for example, the expression *({P^n} f a),* where *f* and *a* are both functions. The original copying strategy - i.e. *({P} f a)* -,would copy the whole expression *(f a),* whereas the new one merely copies *f*. The argument expression *a* stays at its original location. However, the *result* of *a* can be copied later if evaluation of *f* requires it. If so, *a* will first be reduced to RNF, and then the result can be copied for as far as it is in normal form.

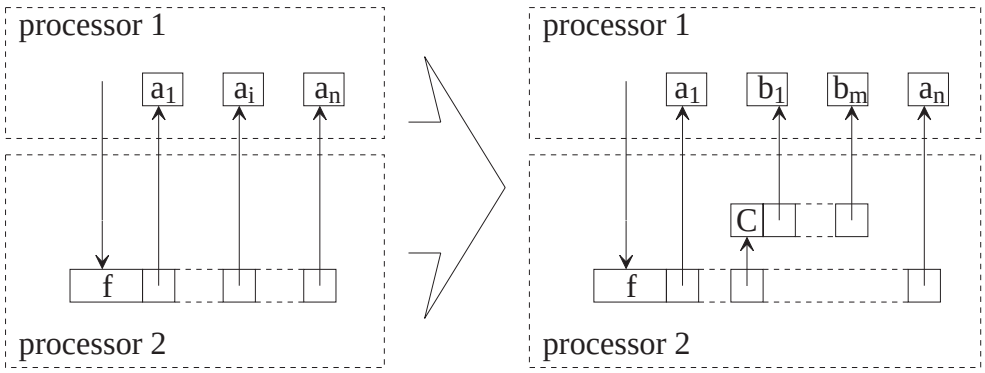


Figure 6-5: *a_1, a_2 ... a_n are all graphs that are not in root normal form. The leftmost picture shows the distribution of graphs after evaluating the expression ({P^n} f a_1 a_2 ... a_n) at processor 1. f is the only function node that has been copied to processor 2. If it needs argument a_i, it will send a request for it. The graph copying mechanism will then start a new process on a_i and return the result as soon as it has been computed. Suppose this is a constructor C with arguments b_1 b_2 ... b_m that are not in root normal form. Only the constructor will then be copied as shown in the rightmost picture. If some b_j is needed later, the same will apply as for a_i, etc. Note that if a_i was already in root normal form before copying f, the rightmost graph would have been obtained directly.*

Again, copying will stop at any function node contained within the result. Consequently, this new copying strategy only copies normal forms by default, and remote processes are able to drive a local computation in a lazy manner and vice versa (hence the name: lazy normal form copying). Figure 6-5 will clarify this behaviour in a more general and a more graphical way.

6.4.2. Safety checks for copying annotated function nodes

The annotated function nodes that do get copied, need to undergo a safety check. Criteria for copying function nodes have been indicated above already. We will not try to exploit all possibilities, but instead we have chosen to keep the copying decision safe and simple (for ease of reasoning about programs). An annotated function node may only be copied if either of the following rules apply.

- It has no unique arguments.
- It has unique arguments, but the function node becomes garbage after copying.

For annotated graphs this is not hard to detect. There are merely two cases we need to consider. Either the annotated function is a locally created function node, or it is not.

1. If the annotated root node is a locally created function node these checks are trivial. It is actually rather hard to create an invalid expression if we annotate a locally created function node. In such a case sharing the annotated graph (as in *(a,a) where a=**{P^n}** f*) does not lead to any problems, as then the *copy* will be shared *after* copying and not the original on beforehand . One would have to place a label between the ***{P^n}*** annotation and the annotated expression in order to share the original (using *(a,b) where b=**{P^n}** a, and a=f*). Such an expression is not very useful in general, as it explicitly copies exported work. If necessary, one could still consider replacing the original node *f* by an equivalent version that does not require unique arguments.
2. If the root is not locally created, it will have been passed as a parameter and the compiler should inspect the uniqueness properties of the passed argument with respect to the current function. If the argument does not have the unique type attribute, copying must be rejected, as then one cannot be sure it becomes garbage. One could consider this an invalid coercion of types. In contrast, if the argument does have the unique type attribute, it may be copied, provided that it is not shared within the function itself. So, for a unique argument the same rules apply as for a locally created function node (see 1.)

A disadvantage of this method, is that the compiler will sometimes refuse copying function nodes that are actually not shared, but lack the unique type attribute. In many cases however, one will not be inclined to start a new process on a (non-unique) graph, mainly because it is often difficult to derive whether it has been - or is being - evaluated already. But if there is need to do so, one could resort to the use of curried functions. This will be explained later in this chapter.

6.4.3. The runtime semantics

This new copying strategy not only avoids uniqueness conflicts, but it also provides clear runtime semantics. Lazy Normal Form Copying avoids the implicit - and conditional - copying of work. Without special measures one can be sure that only normal forms are copied. If one explicitly copies a function (node) *f* to another processor to deliver some structure in parallel, one can be sure that the structure will be returned completely evaluated and not some function that computes (parts of) it. The same holds for the arguments passed to *f*: these will be evaluated locally so that the remotely evaluating *f* will get evaluated arguments and not extra work. One does not risk to copy more work than intended. In effect, the type of an annotated graph indicates what will be copied eventually. With the original lazy copier a similar copying behaviour could only be approximated by using extra annotations. In section 6.6, we will see that lazy normal form copying does not result in an awkward style of programming.

Not only programmers benefit from having clear runtime semantics. The compiler does as well. Typically, a compiler employs strictness analysis to introduce eager evaluation where appropriate. Unfortunately, this may cause a serious side-effect if function nodes can be copied implicitly: changing the order of evaluation might then influence the location that functions are evaluated. If a function is evaluated before copying, it will be evaluated locally, otherwise it will be evaluated at another processor. Therefore, the Concurrent Clean system does not yet derive strictness information for expressions that have an annotation for parallelism. The new normal form copying strategy avoids these problems. It allows the use of strictness analysis and strictness annotations without influencing the location of evaluation.

6.4.4. The effects on efficiency

In addition to clear runtime semantics, lazy normal form copying does not hurt performance. We have tested the parallel programs of table 5-1 with both the old and the new copying strategy and found no significant differences in the speed of execution. The remaining chapters will show some additional parallel examples that perform very well with lazy normal form copying.

Note furthermore that lazy normal form copying allows the introduction of additional deferred nodes within data structures (see section 6.2.4). One merely needs to inject extra functions, such as the indirection function *i*, which is defined as $i\ x = x$. Such a function does not alter its argument data structure, but, as it is represented by an ordinary - and thus deferred - function node, it acts as a copy-stopper. In this way the size of copies (i.e. data) can be limited in a very flexible way, either by hand, or automatically. We exploited this possibility in both chapter 7 and 8.

And finally, note also that lazy normal form copying eliminates the need for expensive mechanisms that avoid the copying of work (reconsider section 5.3.4). By default, work does not get copied at all; it sticks to its location.

6.5. Copying of work using lazy normal form copying

Lazy normal form copying also has a drawback: copying of work becomes more difficult. Suppose one has an argument *a* and one wishes apply *f·g* to *a* at another processor. The ***{Pⁿ}*** annotation only copies the function node at the root, so that ***{Pⁿ}*** *f (g a)* means that only *f* is computed at the other processor, while *g* is evaluated locally - either on beforehand if *g* is evaluated eagerly, or later if it is evaluated in a lazy manner. Some extra work is needed to specify the demanded behaviour. For instance, one might define a new function *h*, which is defined as *h a = f (g a)*. ***{Pⁿ}*** *h* a would now perform as intended.

The introduction of such functions seems to be problematic in certain cases. Take the example in the frame below, where *h* could be some skeleton for parallelism (see also chapter 8). The argument function *g* will not be evaluated another processor, together with *f*. Suppose it should. How does one accomplish this? Perhaps an extra annotation would be needed. We will take a look at this next and show that this is not necessary, as the use of curried functions provides a powerful means to copy work explicitly. But before we proceed, we need to understand how currying is realised in Clean.

```
h :: (x -> y) x -> z
h g a = {Pⁿ} f (g a)

f :: y -> z
...
```

6.5.1. Currying in Concurrent Clean

Clean is based on a functional term-graph rewriting system. In such a system, all symbols have a fixed arity. If a function has non-zero arity we cannot use the function symbol all by itself, thus preventing the use of functions as an argument or as a result. Notwithstanding, we can simulate the concept of higher order functions as follows.

Suppose we have defined a dyadic function *f x y = ...* and we want to model the function *f x = λy.f (x, y)*. We can define a *curry variant* F_1, which is a constructor of arity 1 (on the abstract ABC machine level this is also known as a partial application node). The expression F_1 *x* will now represent *λy.f (x, y)*. Note that the curry variant is in root normal form. To be able to apply it to additional arguments, we also define an *application rule* for the special function symbol *ap*:

$$ap :: (a \rightarrow b)\ a \rightarrow b$$
$$ap\ (F_1\ x)\ y = f\ x\ y$$

In general, one can define for each function symbol *f* of arity *n* a set of *n* curry variants and application rules.

$$
\begin{array}{ll}
ap :: (a \to b)\ a \to b & \\
ap\ F_0\ x_1 & = F_1\ x_1 \\
ap\ (F_1\ x_1)\ x_2 & = F_2\ x_1\ x_2 \\
 & \ldots \\
ap\ (F_{n-1}\ x_1\ x_2 \ldots x_{n-1})\ x_n & = f\ x_1\ x_2 \ldots x_{n-1}\ x_n
\end{array}
$$

In Clean, the curry variants and the application rules are introduced implicitly for each function. To make it possible to deal with the curry variants in a more natural way, Clean allows the curry variant $F_m\ x_1\ x_2 \ldots x_m$ to be denoted as $f\ x_1\ x_2 \ldots x_m$ (where m is smaller than the arity of the function symbol). Likewise, to specify the application of an argument a to a curried function g one can simply write $g\ a$ instead of $ap\ g\ a$.

Thus, curried functions are actually constructors with a number of function arguments. Consequently, the uniqueness type system treats curried functions (almost) like constructors. It ensures that uniqueness properties propagate upwards. Suppose that the m^{th} argument of a function f gets the uniqueness type attribute. According to the standard rules, the curry variants themselves have to be unique in order to ensure uniqueness of the m^{th} argument. The application rules for f become as follows. We have included the type of each application function to be able to indicate uniqueness type attributes by means of the star symbol. The star at the actual arguments more precisely indicates where uniqueness is required.

$$
\begin{array}{ll}
ap^{(a\to b)} :: (a \to b)\ a \to b & \\
ap^{(a\to b)}\ F_0\ x_1 & = F_1\ x_1 \\
 & \ldots \\
ap^{(*a\to b)} :: (*a \to b)\ *a \to *b & \\
ap^{(*a\to b)}\ (F_{m-1}\ x_1 \ldots x_{m-1})\ *x_m & = *F_m\ x_1 \ldots x_{m-1}\ *x_m \\
 & \\
ap^{*(a\to b)} :: *(a \to b)\ a \to *b & \\
ap^{*(a\to b)}\ (*F_m\ x_1 \ldots *x_m)\ x_{m+1} & = *F_{m+1}\ x_1 \ldots *x_m\ x_{m+1} \\
 & \ldots \\
ap^{*(a\to b)} :: *(a \to b)\ a \to *b & \\
ap^{*(a\to b)}\ (*F_{n-1}\ x_1 \ldots\ *x_m \ldots x_{n-1})\ x_n & = f\ x_1 \ldots\ *x_m \ldots x_{n-1} x_n
\end{array}
$$

This means we basically have four classes of application functions, say $ap^{(a\to b)}$, $ap^{(*a\to b)}$, $ap^{*(a\to b)}$, *and* $ap^{*(*a\to b)}$, that require different uniqueness properties for their two arguments. When applying an argument to a curried function, one of these *ap* functions will be inserted implicitly, according the actual uniqueness properties of the curry variant and the required uniqueness properties of the argument. However, if a function f requires a unique argument, some of its application rules will be defined for unique curry variants only: there is no rule $ap^{(a\to b)}\ (F_{n-1}\ x_1 \ldots\ *x_m \ldots x_{n-1})\ x_n$ in the set of rules above. This means that using $ap^{(a\to b)}$ for applying a non-unique curry variant of f to an argument would result in an irreducible (i.e. non-matching) expression. To maintain referential

transparency we need to avoid this situation (from a programmers point of view there is no difference between $ap^{(a\to b)}\ f\ a$ and $ap^{*(a\to b)}\ f\ a$: it all looks like $f\ a$). Therefore, the type system will not allow coercion of unique curry variants to non-unique ones. Unique curry variants are *essentially unique*, and consequently, data structures that contain unique curry variants are essentially unique as well. The use of such objects is severely restricted.

In short, curried functions are treated in a very special way by the type system. With respect to uniqueness, they act more like constructors, than like function nodes. The type system ensures that uniqueness propagates upwards for curried functions.

6.5.2. Currying and copying

Back to the copying of work. The use of curried functions forms an interesting idea in this respect. The type of a curried function argument is denoted as $(A_1\ A_2\ ...\ A_n \to R)$. If the graph copier encounters an object with this type and if the type should indicate what gets copied, one could argue that a function should be copied that takes arguments of type A_1, A_2, ..., A_n, and delivers a result of type R. In contrast, one could also decide not to copy the curried function as it represents work just as well as ordinary functions. We have chosen for the former solution as it provides a clear and powerful way to copy work safely. In addition, it keeps the standard evaluation mechanism simple: if one cannot copy curried functions, one has to incorporate an additional mechanism that applies local arguments to remote curried functions (instead of copying the curried function to the argument and applying it locally).

As we have seen above, uniqueness propagates upward for curry variants. As a consequence curry variants can be copied safely, as opposed to function nodes. As one might expect, this can be realised very easily in Clean. The low level representation of curried functions is the same as the representation of constructors. This allows the new copying strategy to automatically handle curried functions the right way, that is to copy them.

Doing so, we can safely transmit work, while having the additional advantage that $\{P^n\}\ ap\ f\ x$ means exactly the same as $\{P^n\}\ f\ x$ (that is, in both cases the function f will be executed at another processor, even though f is not the root node in $ap\ f\ x$). In a way, the graph copying mechanism presented in chapter 5 transforms $\{P^n\}\ f\ x$ into $\{P^n\}\ ap\ f\ x$, as it replaces the code of the function node f by the corresponding function descriptor before transmission. Consequently, one could argue that at some level *ap* is the only function node that remains copyable. Note also the relation between the copying rules for the root node and the uniqueness properties of curried functions.

Using currying to pass work to other processors, the example of the previous subsection becomes as follows. Here we can see that the type of the annotated function h' indicates that work will be copied. It has type $(x \to y)\ x \to z$, in contrast to f, which has type $y \to z$.

```
h :: (x -> y) x -> z
h g a = {Pn} h' g a

h' :: (x -> y) x -> z
h' g a = f (g a)

f :: y -> z
...
```

A special case is formed by functions of arity zero. One cannot use the currying mechanism to pass such functions to other processor, simply because one cannot use them in a curried way. It remains to be seen whether this is a serious problem. A possible solution would be to introduce additional functions with dummy arguments. Alternatively, one might introduce special language constructs to deal with this problem.

6.6. The runtime semantics of some example programs

Due to the different copying strategy the runtime semantics of Concurrent Clean programs will change. The previous examples already made this clear. This section will show how some rather basic example programs are affected. Note that no well-defined copying strategy can alter the final outcome. It is merely able to influence runtime behaviour and thus, efficiency.

6.6.1. Nfib

This is a notorious benchmark. A parallel version can be defined in Concurrent Clean as follows.

```
nfib 0 = 1
nfib 1 = 1
nfib n = 1 + (nfib (n - 1)) + ({Pn} nfib (n - 2))
```

The difference between the old and the new copying strategy can be clearly seen in this example. The original copying strategy - using ***{P}*** *nfib (n - 2)* - would cause the whole expression *nfib(n-2)* to be sent to another processor by means of a single message, which is very efficient. In contrast, the new copier will keep the argument *(n-2)* local and it will only be evaluated (and copied) after the nfib function requests its remote argument. This clearly is less efficient, due to the delays involved in accessing the argument. To re-obtain the behaviour of original copying strategy one could use an intermediate function *nfib'* as show below.

```
nfib 0 = 1
nfib 1 = 1
nfib n = 1 + (nfib (n - 1)) + ({P^n} nfib' n)
  where nfib' n = nfib (n - 2)
```

Another efficient solution would be the use of a strictness annotation as depicted below. This will force *(n-2)* to be evaluated before creating a new process.

```
nfib 0 = 1
nfib 1 = 1
nfib n = 1 + (nfib (n - 1)) + ({P^n} nfib !(n - 2))
```

A strictness annotation is needed above, because the Concurrent Clean system does not yet derive strictness information for expressions that have annotations for parallelism. This lack of strictness analysis may seem odd, but consider the side-effects that strictness analysis would cause using the original copying strategy: it could change the location that functions are evaluated (see also section 6.4.3). The new copying strategy does not have this problem: the location that arguments are evaluated does not change by placing a strictness annotation, nor by employing strictness analysis.

6.6.2. The sieve of Erathostenes

Another advantage of lazy normal form copying surfaces when using some form of (lazy) stream processing. The sieve of Erathostenes is a well-known example. The original copying strategy required the programmer to place annotations for two reasons. First of all to drive computation, and secondly to defer copying at certain function nodes. This can be seen in the example below, where an ***{I}*** annotation is needed at a filter function to keep it at the current processor when a new parallel sieve is started. This is a rather awkward use of the ***{I}*** annotation.

```
sieve [p : s]   = [p : {P} sieve {I} (filter s p)]
sieve []      = []

filter [x : xs] p
    | x mod p == 0 = filter xs p
    | otherwise    = [x : {I} filter xs p]
filter [] p        = []
```

In contrast, if one replaces the ***{P}*** annotation above by a ***{P^n}*** annotation the first ***{I}*** annotation is not needed anymore. The filter function will then remain at the correct location. A new process will automatically - and lazily - be started on the filter function if

its result is needed by the next sieve. If one does not care about speed of execution and just wants to evaluate the sieve in a distributed lazy manner, no ***{I}*** annotations are needed in the filter function either. In the next chapter we will see how we can improve the sieve program further.

6.6.3. A simple divide and conquer program

And finally, consider the following divide and conquer program to count the occurrence of some element in a tree.

```
count:: a (Tree a) -> Int
count elem NilTree = 0
count elem (NodeTree elem' left right)
    | elem == elem' = count_sons + 1
    | otherwise     = count_sons
    where
        count_sons = ({P}count elem left) + ({P}count elem right)
```

Here we can see the danger of implicitly passing extra work to each processor that evaluates a count function in parallel. The original copying strategy copies function nodes that are contained in the sub-graphs, for instance elements that have not yet been computed. These may represent a substantial amount of work, and if they are shared this work will be duplicated. However, this extra amount of work does not show in the program above. Thus, locality of reasoning is lost with the original copying strategy.

To avoid this problem using the original copying strategy, one had to evaluate the tree explicitly - using annotations - before passing it to the counting function. The new copying strategy does not have this problem. Consequently, it allows programmers to determine locally what work will be performed by another processor.

But suppose one wants to evaluate the elements of the tree at the processor that executes the count function. How does one accomplish this with the lazy normal form copying strategy? Obviously, it is possible to define a function that both creates a tree and performs a count on it, but how do we proceed when we have to deal with an existing tree, of which some elements have been computed and others have not. In that case, we need construct the tree differently. We need to put special functions in the tree. Again, we will not invent extra annotations, but resort to the use of currying. If the elements of the tree must be computed at another processor, the type of the tree must be changed. One could use the following example.

```
:: Tree x = NodeTree ((Tree x) (Tree x) -> x) (Tree x) (Tree x)
          | NilTree
```

The curried function indicates the function that should be applied to both sub-trees to get the element of its root node. The type of this curried function is rather arbitrary though.

It could have been defined having a different type of argument or a different number of arguments (but not zero). The count function would now become as indicated below. Computing the element has become explicit. The definition of the curried function precisely states what will happen if it is evaluated at another processor (the programmer *has* to take care of this now). Note that the arguments of the curried function will not necessarily be evaluated at the same processor as the curried function itself. For these the same rules apply as for annotated functions.

```
count elem NilTree = 0
count elem (NodeTree function left right)
    | elem == function left right = countrest + 1
    | otherwise                   = countrest
    where
        countrest = ({Pn}count elem left) + ({Pn}count elem right)
```

In some cases it will be more efficient to perform such a count function on a graph that has been distributed over a network already. The count function should then be started at the location of its argument. In this case there would be no difference between the old and the new copy function as both would merely copy the count function. This cannot be expressed in standard Clean however. Chapter 8 will introduce some special functions that add this functionality.

6.7. Conclusions

Lazy graph copying is an extension of the standard graph rewriting semantics. We have shown that it may conflict with the uniqueness type system. This depends on the copying strategy. Until recently, Concurrent Clean employed a strategy that was incompatible with the uniqueness type system. It could invalidate uniqueness information during runtime. We have presented a new copying strategy that does not have this problem. Additionally, it does not copy work implicitly. This not only allows programmers to reason more easily about the exact behaviour of parallel programs, but it also ensures that placing strictness annotations cannot influence the location of evaluation.

7. The Costs of Graph Copying

This chapter will explore the costs of graph copying in detail. It will make clear that these costs can form a bottleneck for a class of serious parallel programs. This not only comprises practical divide and conquer style programs - of which parallel matrix multiplication is an example -, but also programs that use pipelines, such as the sieve of Erathostenes. One can observe that copying costs vary widely for different data structures. We will show how arrays can be used to reduce copying costs considerably. This resulted in significant speed-ups for the examples above.

Section 7.1 will start with a short introduction. The next section will clarify in which cases communications costs can form a bottleneck. It will indicates three directions of interest for reducing these costs. The following three sections will focus on each direction: section 7.3 will make clear that choosing data structures carefully can greatly reduce absolute communication costs; section 7.4 shows the use - and the limits - of distributing communication tasks; and thirdly, section 7.5 will show how delays can be reduced by overlapping communication and computation. Considerable speed-ups have been obtained in this way. The last two sections will present our conclusions and discuss the solutions we have adopted.

7.1. Introduction

As we have seen before, distributed graph reduction requires transmission of graphs. This chapter will focus on the costs involved. More precisely, we will not concern ourselves with the pure transmission costs, but with costs that surround the act of transportation. First of all, bookkeeping tasks - possibly employing control messages - are needed to regulate transmission (see also chapter 5). And secondly, as we have seen before, efficient reduction requires a graph representation that differs from the one needed for efficient communication. During reduction a node typically contains local memory addresses to efficiently access other nodes on the same processor, whereas a graph that is being transported will contain some sort of globally usable addresses, such as offsets within a message. This introduces the need for graph conversions. As previous chapters have indicated this basically involves graph copying, but in this chapter we will rather refer to this as *conversion*, to distinguish this kind of copying from the one used by garbage collectors. Clearly, administration and conversions take time, as no special hardware is assumed to accomplish these tasks. As we will see below, the overheads can be unacceptably high for some parallel programs.

These protocol costs have been largely ignored by the implementors of parallel functional languages. In the first place, of all implementations, there are not many aimed at distributed memory architectures. Most recent research in parallel functional programming focuses at implementations for (virtual) shared memory architectures (<*v*,*G*>, Augustsson and Johnsson, 1989; AMPGR, George, 1989; GAML, Maranget, 1991; GRIP, Peyton Jones *et al*., 1987, 1989-b; Flagship, Watson *et al*., 1986, 1987, 1988; HyperM, Barendregt *et al*., 1992). Graph copying plays no role in such implementations. And secondly, the exact costs of copying have not been made explicit for distributed memory implementations. Sometimes the use of an interpreter causes uncertainty about the relative costs of copying compared to the costs of computation (PAM, Loogen *et al*., 1989; π-RED+, Bülk *et al*., 1993). In other cases only some simple divide-and-conquer programs have been tested that are not very conclusive with respect to overall communication overheads (HDG, Kingdon *et al*., 1991; SkelML, Bratvold, 1993). In particular pipelines of processes are not considered at all.

Our work has some relation with that of Kuchen (1994). He proposed, implemented, and tested a set of skeletons on special data structures to exploit data-parallelism. This was based on the assumption that some data structures are cheaper than others for certain problems. Amongst others, he considered algorithmic properties of data structures, such as the ability to efficiently access elements in a random way. However, this work is limited to data-parallelism only, and although it mentions the advantages of certain structures with respect to communication, it does not examine the costs of graph copying in detail.

7.2. Copying costs

Reports on speed-ups for parallel implementations of a functional languages on machines with distributed memory are rare and strikingly similar. Either abstract machine code has been interpreted, resulting in performance figures that are not conclusive with respect to the possible performance of compiled code, or one does compile, but then only a small set of programs turns out to perform well. These programs all have a comparable structure that makes them less sensitive to delays in communication.

A closer look at this sort of program reveals a number of similarities. In the first place, they all exploit a simple divide-and-conquer mechanism. Computation unfolds into a tree of processes. These trees are not very deep, but they contain a huge number of processes. This means that data does not have to travel far. At the same time there will be many processes per processor, so that a waiting process does not imply the processor becomes idle. In addition, the complexity of the data is rather small, compared to the complexity of the computation. As a result, communication overheads can easily be ‘suppressed’ by increasing the problem size. Altogether, this means that communication latencies do not (need to) have much effect on efficiency. Typical examples are programs like nfib, queens, mandelbrot, and ray tracers (see table 5-1).

Let us assume that we do not have such a divide and conquer program. Data will be complex, or it will have to travel over a long distance, or both. We will see realistic examples below. The price of copying cannot be ignored now, it has to be kept as small as possible.

There are basically three ways to do this. First of all, one should decrease the absolute costs of conversion and communication. Secondly, communication and computation should overlap as much as possible, hiding delays. And finally, copying tasks should be distributed over the network as much as possible. The latter can usually be realised by selecting a suitable algorithm for splitting up the problem. We will give a small example of this, but the main emphasis will be on improvements of the first and second kind. These are somewhat more generally applicable than distributing copying tasks, as this strongly depends on the algorithm used. The following section will be concerned with reduction of the overall conversion costs (transmission costs are mostly determined by the hardware, so we will not consider them here). Hereafter, we will shortly contemplate the possibility of distributing graph copying. And finally, we will focus on overlapping computation and communication.

7.3. Decreasing conversion costs

Conversion costs are most significant if data is complex. Large data structures can impose serious overheads when copied to another processor. An example matrix multiplication program will clarify this. Comparing different data structures below, we will see that it is important to choose data structures carefully, because the conversion costs may vary tremendously for different ones. This section will end with a short reflection on distributing copying tasks over the network.

7.3.1. Matrix multiplication with lists

Matrix multiplication is more complicated than the simple divide and conquer programs of the previous section, because the arguments and results are more complex. Increasing the problem size will not automatically result in better performance as the copying costs will grow as well. Depending on the size of the network the complexity of communication may approach the complexity of plain matrix multiplication: $O(n^3)$. This happens for example when every new matrix element is computed by a separate processor: each processor needs $O(n)$ data and $O(n^2)$ processors are needed. It is crucial that copying costs are kept to a minimum here.

A simple way to multiply two square matrices A and B of size n (number of rows) on n processors, is to split A into n rows $a_1 ... a_n$ and compute $a_i \times B$ on processor p_i. We will assume that A and B both reside at some processor p_S initially and that the result should return to p_S. Suppose further, that p_S itself distributes all the work. Below, we have listed a Concurrent Clean program that multiplies two matrices in this way.

```
mul :: [[Real]] [[Real]]  -> [[Real]]
mul a b = mul' a (!Transpose b) processor1
```

continues

continued

```
where
    mul' :: [[Real]] [[Real]] Int -> [[Real]]
    mul' [a1:a] b p
        = [{P at (ItoP p)} mulrow a1 b : !mul' a b (p + 1)]
    mul' [] b p = []

    mulrow :: [Real] [[Real]] -> [Real]
    mulrow a1 [b1:b] = [!mulvector a1 b1 : !mulrow a1 b]
    mulrow a1 []     = []

    mulvector :: [Real] [Real] -> Real
    mulvector a1 b1 = mulvector' a1 b2 0.0

    mulvector' :: [Real] [Real] Real -> Real
    mulvector [e1:r1] [e2:r2] result
        = mulvector r1 r2 (result + (e1 * e2))
    mulvector [] [] result = result
```

It is easy to see that this will not lead to speed-ups if communication is expensive: copying B (n-1) times is $O(n^3)$. In general, if we use this method on p processors, the costs of communication are $O(p \times n^2)$. We can see below what results were obtained with matrices represented by lists of lists of floating point numbers, employing up to 16 processors. We should note here that the use of lists does not impose overheads to access matrix elements: the algorithm does not need random access to elements.

Table 7-1: *Multiplication of matrices represented by lists of lists of floating point numbers.*

	sequential	16 processors	speed-up
multiply 64×64	3.9 sec.	5.0 sec.	$\times$ 0.8
multiply 128×128	30.3 sec.	35.0 sec.	$\times$ 0.9

Table 7-2: *Time spent by root processor in parallel version on conversion and garbage collection (using lists of lists).*

	conversions	collections
multiply 64×64	2.9 sec.	0.2 sec.
multiply 128×128	16.9 sec.	10.0 sec.

The disappointing figures in table 7-1 need some explanation. First of all, the conversion costs per matrix element are higher than an elementary multiplication step.

Conversion costs can be derived from table 7-2 and the following formula, which elucidates the conversion costs at the root processor (using a total of 16 processors).

$$\text{conversion costs at the root: } 15p\left(n^2 + \frac{n^2}{16}\right) + 15u\frac{n^2}{16} \approx (16p + u)n^2 \qquad (1)$$

It assumes that it takes *p* seconds to pack an element and *u* seconds to unpack it. The root (un)packs data for 15 remote processors and *n* denotes the size of the square matrices). We should note that unpacking is less expensive than packing in our implementation, as unpacking does not involve copying nodes. It merely has to adjust pointers. We have not yet established the exact difference in speed. This means that conversion takes between $16pn^2$ and $17pn^2$, assuming that *u* is greater than zero, but smaller than *p*. Consequently, the packing cost *p* lies approximately between 42 and 45 μs for $n = 64$ and between 61 and 65 μs for $n = 128$ (the total conversion costs can be found in table 7-2 for these values of *n*). The variation is caused by extra overheads if memory runs low during conversion, which is the case during multiplication of matrices of 128 by 128 reals. For comparison, from the timings for one processor one can deduct that it takes about 15 μs to perform a basic multiplication step (total time is n^3 times basic time). This is considerably cheaper than conversion.

Secondly, the larger the matrices, the more the tests suffer from excessive garbage collection times at the root processor, as we can see in table 7-2. The list representation of each matrix consumes about 400 Kbyte of memory, while graph conversion claims a similar amount for the resulting message. Only little room remains to perform graph reduction. With a total heap size of 3 Mbyte per transputer, garbage collection times add up to 10 seconds. The root processor spends 27 seconds on conversion and garbage collection alone, while matrix multiplication takes less than 2 seconds per processor. This is partly due to the use of a two-space copying collector, which traverses all live data and limits the maximum usable space to half the heap size. For this particular problem it might be advisable to reduce the number of garbage collections by using an heuristic that will hold garbage collections as long as the heap contains large - or many - messages. Garbage collection would not free enough memory in this case, as opposed to communication. However, this introduces alternating phases of communication and computation, which may not always be advantageous. We have not yet tested this.

And finally, idle time is considerable. A large amount of data has to be transported over the transputer communication links. In addition, conversions and garbage collections cause delays as well. Not only do these operations take notable time to complete, they also - unlike normal reduction - make overlapping computation and communication more difficult. One cannot boldly access messages in the heap during these operations because nodes may be in an inconsistent state (perhaps containing forwarding addresses and marks). In the current implementation processors cannot forward messages while they are busy converting graphs and collecting garbage. The matrix multiplication program suffers greatly from this problem. Again, part of this problem might be alleviated by postponing garbage collections and conversions, in favour of message passing.

From the above it becomes clear that copying is excessively expensive here. In short, converting a large graph takes much time and memory. One can observe that relative conversion costs can be reduced by increasing the problem size, but at a much slower rate than for the simple divide-and-conquer programs listed earlier. At the same time physical memory size restrains the use of extremely large matrices. These introduce substantial memory management overheads, which in turn will cause higher communication delays.

7.3.2. Matrix multiplication with arrays

We will show below that the use of strict arrays solves many problems. In this particular case we do not need them for efficient random access to elements, but because they can be copied cheaply. In Clean, the uniqueness type system allows an array to be represented by a single, efficiently modifiable, block of memory. Strict arrays consist of evaluated (unboxed) elements that are placed one after another in memory. As a result, copying a strict array merely involves moving a block of memory if the elements are flat (e.g. a strict array of integers). No expensive conversion is involved, which can clearly be seen in the tables below. The algorithm is virtually the same as the one above that uses lists. We will only give the new definition of the *mulrow* and the *mulvector* function to illustrate the use of (unique) arrays. The *put* and *get* functions are needed to insert and to extract arrays elements (which Clean 0.8 does not support directly).

```
mulrow :: Array Matrix Int -> *Array
mulrow a1 b row
  = mulrow' a1 b (newArray size) row size
  where size = sizeof a1

mulrow' :: Array Matrix *Array Int Int -> *Array
mulrow' a1 b c row col
  | col == 0  = c
  | otherwise = mulrow a1 b (put1 c col' result) row col'
  where
      col'   = col - 1
      result = mulvector a1 b size col' 0.0
      size   = sizeof a1

mulvector :: Array Matrix Int Int Real -> Real
mulvector v1 v2 row col result
  | row > 0   = mulvector a b row' col result'
  | otherwise = result
  where
      row'    = row - 1
      result' = result + ((get1 a row') * (get2 b row' col))
```

To achieve these results, we had to optimise the copying of arrays. Clearly, if the graph is a strict array with flat elements, it can be copied without conversion by means of a simple block move. In many cases however, the graph will consist of a mixture of strict arrays and ordinary nodes. If an array is small, it will not be worthwhile to send a separate message for it. In this case the graph copier will pack it - i.e. copy it - in a message together with the other nodes of the graph. On the other hand, if an array is large, the copier will create a separate message for it. This message consists of the array itself, creation does not involve copying. For our transputer system 16 Kbyte turned out to be a good boundary between small and large arrays. The combination of two-dimensional strict arrays and this copying scheme has led to the improved speed-ups below.

Table 7-3: *Multiplication of matrices represented by 2-dimensional arrays of floating point numbers.*

	sequential	16 processors	speed-up
multiply 32×32	1.0 sec.	0.4 sec.	$\times$ 2.5
multiply 64×64	7.6 sec.	1.0 sec.	$\times$ 7.6
multiply 128×128	60.3 sec.	5.0 sec.	$\times$ 12.1
multiply 256×256	480.7 sec.	32.5 sec.	$\times$ 14.8

Table 7-4: *Time spent by root processor in parallel version on conversion and garbage collection*

	conversions	collections
multiply 256×256	0.03 sec.	0.08 sec.

If we compare these results with the previous ones, the merits of using strict arrays are clear. Larger matrices can be used and for the large ones the speed-up is nearly perfect. And if we take a closer look, we can see that all problems listed above are avoided by using this different data representation.

- There are nearly no conversion costs, as large matrices do not get packed at all. They already are packed from the start.
- Virtually no time is spent on garbage collections. On the one hand fewer garbage collections are needed: no memory is allocated for conversions and the matrix representation is more compact. A matrix of 128 by 128 reals takes only about 128 Kbyte as opposed to 400 Kbyte above. On the other hand each garbage collection is less expensive, as the amount of live data is smaller, while having a less complex structure as well.
- Idle time is small. Less data has to be transported, due to the compact matrix representation. The reduction of conversions and garbage collections is important as well. Unlike these computations, normal reduction can easily be overlapped by communication, because the transputer communication hardware works independently of the CPU.

However, we can also observe that the absolute execution times on a single processor have gone up. It seems that computing the address of an element in a two-dimensional array is more expensive than walking through a list. Unfortunately, version 0.8 of the Clean compiler does not support arrays directly, so that special functions are needed to perform array selections (version 1.0 solves this problem, but it has not yet been ported to the transputer system). Small functions like *put* and *get* are used to insert or extract array elements. If we manually inline the code of the *get* functions for the inner vector multiplication loop (the *mulvector* function) we obtain the results in table 7-5 for sequential matrix multiplication. These are competitive with C.

Table 7-5: *Execution times of sequential matrix multiplication in Clean and C using two-dimensional arrays and inlined array selections. The figures for the Pact Compiler indicate execution times obtained with the stack in fast on-chip memory (1) and in off-chip memory (2). The latter is more realistic for parallel programs with multiple processes on a single processor. In Clean and in Helios, stacks are always placed off-chip. The Helios compiler produces the best code in this case.*

Matrix Size	Clean	Helios C	Pact C[1]	Pact C[2]
multiply 32×32	0.3 sec.	0.2 sec.	0.2 sec.	0.3 sec.
multiply 64×64	2.3 sec.	1.9 sec.	1.8 sec.	2.2 sec.
multiply 128×128	17.6 sec.	15.7 sec.	14.3 sec.	18.2 sec.
multiply 256×256	139 sec.	131 sec.	117 sec.	146 sec.

The differences between Clean and C are marginal in these case. For Clean we can expect further improvements if loop optimisations are employed that are commonly found in languages like FORTRAN and SISAL. Considering the code that the Clean system currently produces, we expect that performance can be improved approximately by factor of two (compared to version with inlined array selections of table 7-5).

Table 7-6: *Parallel matrix multiplication in Clean using two-dimensional arrays and inlined array selections.*

	sequential	16 processors	speed-up
multiply 32×32	0.3 sec.	0.3 sec.	$\times$ 1.0
multiply 64×64	2.3 sec.	0.7 sec.	$\times$ 3.3
multiply 128×128	17.6 sec.	2.9 sec.	$\times$ 6.1
multiply 256×256	139 sec.	14.9 sec.	$\times$ 9.3

Using the inlined array selector functions we get better performance for the parallel version as well. These figures are listed in table 7-6. We can see that the sequential versions are almost twice as fast as the ones that used lists. On the other hand, not only the execution times have decreased, but the speed-ups have as well. This situation becomes even worse when using lists of one-dimensional arrays instead of two-dimensional arrays. One-

dimensional arrays have the advantage that addressing is cheaper than for two-dimensional arrays. This reduces the sequential execution times notably, and it reintroduces slightly more complex communications. As a result, the parallel execution times increase a little and speed-ups plummet. The results are shown in table 7-7.

Table 7-7: *Parallel matrix multiplication in Clean using lists of one-dimensional arrays and inlined array selections.*

	sequential	16 processors	speed-up
multiply 32×32	0.2 sec.	0.3 sec.	$\times$ 0.7
multiply 64×64	1.5 sec.	0.8 sec.	$\times$ 1.9
multiply 128×128	11.0 sec.	4.4 sec.	$\times$ 2.5
multiply 256×256	84.9 sec.	16.5 sec.	$\times$ 5.9

At this point, the limits of the available bandwidth become clearly noticeable. With matrices of 256×256 elements, we need to transport about 8 Mbyte of data from one processor. This causes delays, which becomes more apparent as the running times get shorter. Running these tests on 32 processors will actually increase execution times notably compared to running them on 16, as then we need to ship 16 Mbyte of data from the root processor. This however, is not a problem that is caused by graph copying. It is inherent to the algorithm used. One might solve it by using some form of broadcasting - which should be able reduce the amount of transported data drastically -, or by using a slightly different algorithm that distributes the data in another way. This will be treated below. It is also possible to use a totally different approach with 'data-parallel' matrices that are distributed from the start. As we will see in the next chapter this can give greatly improved speed-ups.

One could argue that lists are more elegant than arrays. In any case, the use of arrays for certain operations does not rule out the use of lists for others. For instance, on top of the efficient parallel matrix multiplication function for two-dimensional arrays, one can define one for a list representation. Full conversions between the two representations are of $O(n^2)$ If one has n processors, parallel matrix multiplication is of $O(n^2)$ as well. One can take $C^{-1}((C\ A) \times (C\ B))$, where C is the function that converts the list representation into the array representation. Having n processors one does not convert B n times, but only once. In effect the conversion has been moved in front of all copying operations so that these can share the converted result. Another viable way to achieve this might be the use of a special cache for converted graphs so that these do not need to be re-computed.

7.4. Distributed copying

We have seen that absolute conversion costs can be greatly reduced by choosing appropriate data structures. Alternatively, one could reduce the time needed for conversions by distributing copying operations over all processors. This is mostly worthwhile if copying costs are high, as is the case for the matrix multiplication program above that used lists. It requires the root processor to perform most conversions, which turned out to be

very expensive. Conversely, one could use an algorithm that avoids this problem. An easy way to achieve this is to split matrices recursively in half until some basic size has been reached. Matrix multiplication will then be performed on these sub-matrices. Combination of the partial results yields the final one. This gave slight speed-ups for matrices represented by lists (table 7-8).

```
mul' :: [[Real]] [[Real]] -> [[Real]]
mul' a b
  | (sizeof a < threshold) = sequential_multiply a b
  | otherwise              = append top bottom
  where
    top           = {P} mul'' half1 b
    bottom        = !mul'' half2 b
    (half1,half2) = split_horizontal a

mul'' :: [[Real]] [[Real]] -> [[Real]]
mul'' a b
  = append' left right
  where
    left          = {P} mul' a half1
    right         = !mul' a half2
    (half1,half2) = split_vertical b
```

Table 7-8: *Parallel Matrix Multiplication using lists of lists and recursive split-up.*

	1 processor	16 processors	speed-up
multiply 64×64	3.9 sec.	2.6 sec.	$\times$ 1.5
multiply 128×128	30.3 sec.	16.0 sec.	$\times$ 1.9

Table 7-9: *Time spent by root processor in parallel version on conversion and garbage collection (using recursive split-up on lists).*

	conversions	collections
multiply 64×64	0.6 sec.	0.0 sec.
multiply 128×128	4.4 sec.	2.5 sec.

These figures are better than the naive matrix multiplication algorithm using lists, but they are far from optimal, and certainly worse than the results obtained with arrays. Compared to the original list version the root spends 4 to 5 times less time on conversions and garbage collections, but the total execution time has only halved. As conversions are not only executed at the root now, but for a considerable part on other processors as well, we need to take into account these costs. Here problems arise because in Clean graphs are copied as a whole - if they are in normal form - and not on a per-node basis. The

distribution of data takes several phases that do not overlap. In other words, the copying costs - mostly conversion costs - add up over a path from the root processor to a leaf. Details can be found in the frame 'Delays in Matrix Multiplication'.

Delays in Matrix Multiplication

For the naive matrix multiplication solution the total amount of delays is largely determined by the conversion costs at the root. If we ignore the effects of garbage collection and define p and u to indicate the packing and unpacking costs per element, we can use the following formula for overall delays caused by conversion.

total conversion delays:

$$15p\left(n^2+\frac{n^2}{16}\right)+u\left(n^2+\frac{n^2}{16}\right)+p\frac{n^2}{16}+15u\frac{n^2}{16}=(16p+u)n^2 \qquad (2)$$

The conversion costs at the root are given by (1), which is almost the same as (2). For the matrix splitting solution this situation differs. The conversion costs at the root are as follows:

conversion costs at the root: $2p\left(n^2+\frac{n^2}{2}\right)+un^2=(3p+u)n^2$ (3)

It indicates that packing costs per element are now roughly between 37 and 49 μs and between 67 and 90 μs respectively (see table 7-9). This is in accordance with the figures found earlier if we consider that memory management overheads are worst for the largest matrices. The formula for the overall delays is rather different however. If we recursively split the matrices four times, we obtain 16 sub-matrices. The delays for splitting and recombining can then be approximated by the following formulae.

packing delays during splitting: $\left(3n^2+3\frac{n^2}{2}+3\frac{n^2}{4}+3\frac{n^2}{8}\right)p$ (4)

unpacking delays during splitting: $\left(3\frac{n^2}{2}+3\frac{n^2}{4}+3\frac{n^2}{8}+3\frac{n^2}{16}\right)u$ (5)

packing delays during recombination: $\left(\frac{n^2}{2}+\frac{n^2}{4}+\frac{n^2}{8}+\frac{n^2}{16}\right)p$ (6)

unpacking delays during recombination: $\left(n^2+\frac{n^2}{2}+\frac{n^2}{4}+\frac{n^2}{8}\right)u$ (7)

total conversion delays: $(4)+(5)+(6)+(7) \approx (6.6p + 4.7u)n^2$ (8)

Depending on the relative costs of unpacking the matrix splitting solution is only a factor 1.5 to 2.4 better than the original one. This largely explains the limited speed-ups that we measured for the splitting solution.

The conversion costs for the splitting algorithm are too high to improve the simple version that uses arrays. However, we can also combine the two solutions. This resulted in the execution times of table 7-10. For this particular test we did not use two-dimensional arrays, but lists of one-dimensional arrays, because this enabled easy splitting.

Table 7-10: *Parallel matrix multiplication using lists of one-dimensional arrays, inlined array selections, and recursive split-up.*

	1 processor	16 processors	speed-up
multiply 32×32	0.2 sec.	0.4 sec.	$\times$ 0.5
multiply 64×64	1.5 sec.	0.9 sec.	$\times$ 1.7
multiply 128×128	11.0 sec.	3.0 sec.	$\times$ 3.7
multiply 256×256	84.9 sec.	11.1 sec.	$\times$ 7.7

Clearly, this improves the earlier array versions only a little, if it improves them at all. This solution appears more scalable though, because it does not add serious overheads for larger networks. In spite of this, we have not been able to obtain better results with additional tests on 32 processors. So at best, we do not increase execution times by using more processors, unlike the earlier versions. Again, the delays form the limiting factor.

To overcome this problem, one either has to come up with an algorithm that splits up the load more effectively, or one has to avoid the standard method of copying. The former may introduce much work for a particular problem, so this is not always the best way to go. In either case, finding better algorithms falls outside the scope of this thesis, so we will concentrate on different forms of copying. For avoiding delays, the use of streams comes to mind. One would like to break up the copying of a structure in pieces, so that a rather continuous stream of data can be formed in which a pipeline of copying functions works concurrently. This will not reduce the overall copying costs, but ideally it will fold them up so that delays are reduced. We have not implemented matrix multiplication based on this method, because implementing a pipelined copying algorithm for arbitrary structures is rather complex and constitutes too much work. The next section will show how such pipelines can be efficiently implemented for lists of integers. We will focus on the sieve of Erathostenes.

7.5. Overlapping communication and computation

To some extent communication in Clean is lazy. By default results are not transmitted until needed. This has some advantages. It avoids needless communications, suppresses small (expensive) messages, and allows remote results to be referred without copying. The main disadvantage is that it increases delays in obtaining results: if a result is needed, one has to wait until it has been returned. First a request message will be transmitted. After this has arrived, the result will be packed as soon as it has been computed, and finally the obtained message can be returned and unpacked (see chapter 5). Only then the requesting process can proceed. This may have great effect on performance for certain programs.

In addition to the programs above, which suffered to a large extent from the complexity of conversion alone, there is a class of programs that is seriously affected by these delays. In general this is a problem if data is not necessarily complex, but has to travel over a large distance. Many (small) delays will then add up to a considerable amount. In functional languages this travelling distance is closely related to the depth of parallel function composition, whereas the costs of each communication step are related to the structural complexity of arguments and results. Although reduction of conversion costs can be important for reducing delays as well, we will not consider this here. The previous section already has focused on this. In contrast, this section will concern itself with reducing delays by overlapping computation and (more eager) communication.

7.5.1. Streams

Streams consist of - possibly lengthy - compositions of functions. These pipelines can be used to solve real-world problems, but they can be very sensitive to delays. A classical example - albeit not extremely useful - is the sieve of Erathostenes. It generates a list of prime numbers by filtering all non-primes from the list of natural numbers. The filter consists of a pipeline of small filters. Each assumes the first number in the stream to be a prime number. It will first deliver this prime and next remove all numbers from the stream that are divisible by it. The resulting stream is passed to the next filter. This results in the following process structure.

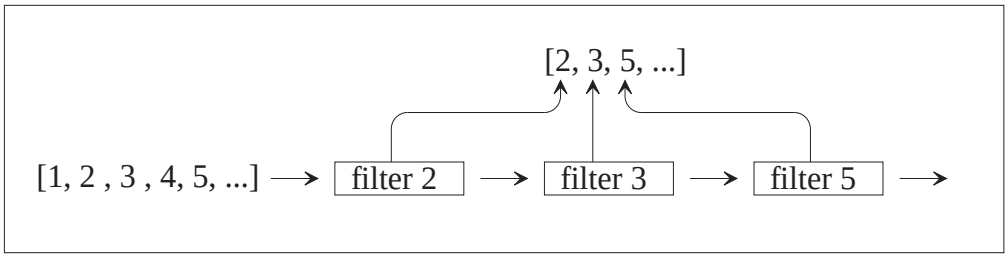


Figure 7-1: *A snapshot of the process structure of the sieve of Erathostenes.*

In Concurrent Clean, this algorithm can be defined as shown below. We have listed the definitions for the most important functions only. New processes are either started at a remote processor with a ***{P^n}*** annotation, or on the current processor with a ***{I}*** annotation. A heuristic has been used to balance the load. The higher the prime number of a filter, the fewer numbers it has to filter, so an increasing number of filter processes will be grouped on the same processor. As processes reduce functions to root normal form only, extra internal processes are needed to drive the filtering (in a speculative way).

The results of this program have been listed in table 7-11. The speed-ups are not very impressive, mainly due to the effects of delay. In addition, the use of the ***{I}*** annotation in the filter function gives rise to unbridled eager computation of results. This may flood the heap, which introduces substantial memory management overheads and prevents the use of large - or infinite - lists. The exact cause of delays varies widely, because there is no mechanism to control the length of the list that will be copied. If lists are long, conversion time introduces considerable delay. If lists are short, many small messages are needed and communication overheads are relatively high.

```
sieve n [p : s]
    | n == 0    = [p : {P^n} sieve ns f]
    | otherwise = [p : {I} sieve (n - 1) f]
    where
        f    = filter s p
        ns   = squareroot p
sieve n [] = []

filter [x : xs] p
    | x mod p == 0 = filter xs p
    | otherwise    = [x : {I} filter xs p]
filter [] p = []
```

Table 7-11: *The sieve of Erathostenes.*

	1 processor	32 processors	speed-up
sieve 10000	18.4 sec.	8.9 sec.	× 2.0
sieve 20000	61.7 sec.	23.7 sec.	× 2.6
sieve 40000	213.3 sec.	64.7 sec.	× 3.3
sieve 80000	752.2 sec.	?	?

7.5.2. Buffering

To overcome these problems an efficient buffering mechanism between processors is useful. Firstly, it is able to limit the number of elements that can be computed in advance so that memory problems are avoided. Secondly it can minimise delays by controlling the size of messages and the use of early requests, meaning that request messages are sent before the requested object is actually needed. This will cause conversions and communication to take place at less critical moments. Reducers do not have to stop (that long) performing useful computations, so that communication and computation will overlap. By controlling the message size one avoids messages that are too large or too small to be efficient.

Strict arrays are very useful for implementing a buffering skeleton. As has already been shown above, strict arrays can be communicated very efficiently, so buffers implemented this way can be as well. In addition, the uniqueness type system can ensure that buffers are not shared, so that they can be destructively updated. This is very important for efficiently filling and emptying a buffer.

We have implemented the following low level buffering function for lists of integers, which has been based on this method. The definitions below are a little more complicated in reality. For clarity, we have left out boundary conditions and the definitions of low level functions. Each *buffer* function starts a *write* function at the given processor and passes the result to a local *read* function. The *write* function will fill a newly created buffer with the values it obtains by evaluating *size* elements of its parameter list. It finally combines the resulting buffer and a reference to the next *fillbuffer* function for the rest of the list.

Evaluation of the *read* function will force this result to return. It first extracts the returned buffer and the reference to the next one. Hereafter, it sends a request for the next buffer without suspending itself (by starting an interleaved process on it). This forces speculative evaluation of the next *size* elements. Meanwhile, each low level *get_element* function takes the next element out of a buffer. It delivers both the element and the updated buffer. If executed on a depleted buffer it will first perform a *read* on the next one.

```
:: Buffers = (!Array, Buffers)

buffer :: ProcId Int [Int] -> [Int]
buffer processor size list = read({Pn at processor} write size list)

write :: Int [Int] -> Buffers
write size list = fillbuffer (createbuffer size) size list
where
   fillbuffer buf size list = (filledbuffer, nextbuffer)
   where
      nextbuffer = fillbuffer (createbuffer size) size remaininglist
      (filledbuffer,remaininglist) = eval_and_fill buf size list

read :: Buffers -> [Int]
read (filledbuffer, nextbuffer) = [first : get rest {I}nextbuffer]
where
    (first,rest) = get_element filledbuffer

    get buf nextbuffer
        | notempty buf = [elem : get rest nextbuffer]
        | otherwise    = read nextbuffer
                         where
                             (elem,rest) = get_element buf
```

Most of the time the situation of figure 7-2 will exist. Any consumer will be able to get the next element from a returned buffer quickly by evaluating the *get* function. At the same time the *eval_and_fill* function will be filling the next buffer. On average *size* elements will be buffered. If processes proceed at equal speed the next buffer will be filled - and possibly returned - by the time the consumer needs it. Note that evaluation of the list will stop if the consumer does not evaluate the *read* function for some buffer: the next buffer will not be requested then.

As we will see below, the buffer function above can give greatly improved performance for streams of integers. It has two limitations however. First of all, it unravels the argument list, so it does not preserve cycles. Unfortunately, we cannot avoid this if we use Clean to construct the buffer array. In Clean, there is no way to detect cyclic structures. And secondly, this buffer function can only handle lists of integers. This has to do with the

functions that fill the strict array, which need to know the size of each element. This limitation can be avoided in version 1.0 of the Clean system, which supports type classes (the transputer implementation only supports version 0.8).

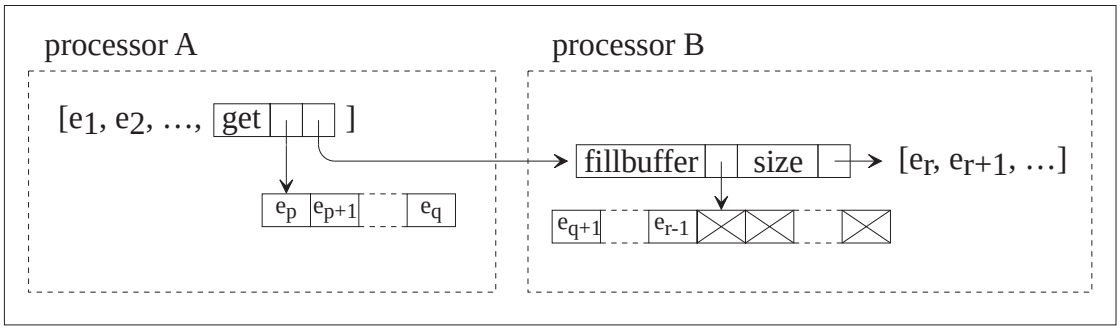


Figure 7-2: *The relation between buffering functions during evaluation of the fillbuffer function.*

If we use this new buffer function for the implementation of the sieve program, we get the code below. It does not need extra process annotations to drive the filtering processes. In order to start up a buffer function that reads from the correct processor each remotely started sieve receives the processor number of its parent. Note that the *currentP* function is not really referentially transparent: it has a different result if evaluated on different processors. On the other hand it cannot have any influence on the final outcome of the program, except for the placement of processes and as a result of this, on efficiency. In addition, as we will see in the next chapter, it is possible to define more sophisticated constructs that do not need such functions. Note also that the copying strategy ensures that the *currentP* function is reduced at the correct processor. It does not need a strictness annotation (which would have been necessary in the old copying strategy).

```
sieve n [p : s]
    | n == 0    = [p : {Pn} psieve currentP ns f]
    | otherwise = [p : {I} sieve (n - 1) f]
    where
        f  = filter s p
        ns = squareroot p
sieve n [] = []

psieve processor n list = sieve n (buffer processor BufferSize list)

filter [x : xs] p
    | x mod p == 0 = filter xs p
    | otherwise    = [x : filter xs p]
filter [] p        = []
```

The execution times for this program have been listed below. Clearly the speed-ups for the sieve are significant if enough numbers are filtered. We should note that sieves are not necessarily running on neighbouring processors and that the buffering function is not yet optimal. Idle time is approximately 50% on each processor. This means that even better performance may be achieved in the future, although these results already are a notable improvement compared to results presented earlier for functional programs.

Table 7-12: *The sieve of Erathostenes with buffering. The first column is the same as in table 7-11, as it lists the results for a purely sequential program, without any buffering function.*

	sequential	16 processors	speed-up	32 processors	speed-up
sieve 10000	18.4 sec.	3.9 sec.	× 4.7	3.5 sec.	× 5.3
sieve 20000	61.7 sec.	9.8 sec.	× 6.3	8.3 sec.	× 7.4
sieve 40000	213.3 sec.	29.8 sec.	× 7.2	21.8 sec.	× 9.8
sieve 80000	752.2 sec.	90.9 sec.	× 8.3	56.4 sec.	× 13.3
sieve 100000	1136.9 sec.	131.4 sec.	× 8.7	81.1 sec.	× 14.0
sieve 200000	4225.8 sec.	476.0 sec.	× 8.9	269.6 sec.	× 15.7

7.6. Conclusions

We have concentrated on the costs of graph copying. These costs turned out to form a bottleneck for a class of useful parallel programs, such as matrix multiplication and programs that use pipelines. Observing that copying costs vary considerably for different data structures we have been able to present a solution that relies on the use of arrays. These can be transferred very efficiently, and they form the basis of efficient skeletons to regulate communication. This has led to significant speed-ups for the example programs.

7.7. Discussion

One may argue that we have actually avoided graph rewriting by using arrays. Does this mean that we have chosen the wrong computational model for Concurrent Clean? And if not, does the use of arrays demand an unnatural way of programming?

The way we have used arrays in this chapter can be compared to the way that files are used by the Concurrent Clean I/O system. In essence, files and messages are the same. Using the standard graph copying mechanism, graphs not only can be stored in a message, but also in a file. Communication can then take place by passing this file from one process to another (à la UNIX). The origins of the flat ‘structure’ of files are found in the nature of the underlying hardware, which only handles flat blocks of data well. Likewise, our use of arrays has to do with communication hardware that only handles flat messages well.

In both cases, flat data structures are used at a low level. At higher levels, graph rewriting remains dominant. This can be seen in the pipelining example above. The buffer function consumes lists and produces lists, but it uses arrays for its implementation, simply

to efficiently interface with the underlying communication hardware. It avoids the standard graph copying mechanism and replaces it by a custom one. In general, the use of arrays can be embedded in ordinary graph rewriting.

For other programs that have nothing to do with communication or other forms of I/O, the use of arrays can be important as well. Only now, the need for such flat structures will usually originate in the algorithm itself. For example, it may be necessary to provide constant access time to elements.

One may argue that it is odd to use arrays in a parallel implementation, as it is necessary to sequentialise the write access on arrays, so that each following function uses the new array that the current function delivers (leading to an imperative style of programming). The uniqueness type system enforces this order. However, lists may not be much better in this respect. These automatically introduce sequential access to elements because of the way they are constructed. One way or another, one ends up with some form of sequentialisation.

Note also that, after choosing a flat data structure, this decision can sometimes be hidden at a higher level. On the language level, ‘list’ comprehensions can be defined for both arrays and lists. For example, the matrix multiplication program can be expressed very concisely with list comprehensions. Both the version that uses lists and the one that uses arrays essentially have the following structure.

```
mul a b       = [mulrow x b \\ x <- a]
mulrow a b    = [mulvector a x \\ x <- b]
mulvector a b = sum [x * y \\ x <- a & y <- b]
```

Such constructs are easy to use and they can safely incorporate parallel updates on arrays. They are related to the language constructs of data-parallel languages like SISAL. We can take this a bit further. It is conceivable that special constructs are provided that enable the efficient splitting and recombination of unique arrays (without copying arrays physically), allowing programmers to define processes that work on all parts in parallel. As we will see in the next chapter, skeletons can be defined on such split arrays that allow a very straightforward use of arrays in a parallel context. So even if we explicitly use arrays, this does not necessarily affect ease of parallel programming.

8. Constructing Skeletons

Skeletons are well-suited to structure parallel programming. They allow the easy use of some well-known parallel programming paradigms to construct portable, efficient programs. Much research has been focused on the use of skeletons in functional programming languages, because they can be expressed elegantly as higher order functions. On the other hand, little attention has been paid to an elementary weakness of skeletons: how to implement them without having to resort to low level techniques. In this chapter we will show that the parallel constructs of Concurrent Clean can be used to efficiently implement a range of high level skeletons. We will construct skeletons for data parallelism, for parallel I/O, and for stream processing. Our experiments demonstrate that no performance penalty needs to be paid, compared to more restrictive solutions.

Section 8.1 will give a short introduction on skeletons. Hereafter, in section 8.2, we will introduce some basic functions that we will use in the remaining part of this chapter to construct a number of skeletons. Section 8.3 will show how to construct skeletons for data-parallelism. In particular, it will focus on a data-parallel matrix multiplication program. In section 8.4 we will demonstrate how skeletons can be constructed that provide a powerful mechanism for specifying parallel I/O. Section 8.5 will proceed with the introduction of skeletons that support stream processing. And finally, section 8.6 will present our conclusions.

8.1. Introduction

The use of functional languages partly solves the problem of writing efficient parallel programs, because referential transparency allows arbitrary expressions to be evaluated in parallel, without changing the outcome of the program. This makes reasoning about the result of parallel functional programs as easy as for sequential ones. Unfortunately, knowing we cannot compute the wrong result, does not imply we are doing it efficiently. Resource allocation has great effect on parallel performance. Finding the best one for a certain program on a given parallel machine model is difficult. In addition, as there are so many different models, portability is hard to maintain.

Should the compiler allocate resources implicitly, or is it a programmers task to do it explicitly? Clearly, a fully implicit approach would be ideal. It would keep the programmer from making mistakes and it retains portability. However, at this moment no method is

known that will automatically derive efficient parallel programs in all circumstances. This means that compilers currently need some form of guidance from the programmer.

The concept of skeletons (Cole, 1989) forms an interesting idea for structuring this guidance. Skeletons can be seen as predefined templates that are used to control parallel execution of programs. The use of these has the advantage that it enables the programmer to exploit certain well known parallel programming paradigms, without having to resort to low level language constructs. In addition, the implementor of a skeleton is able to construct the most efficient one for each platform by taking full advantage of specific machine features. And finally, a certain degree of portability is ensured if the same set of skeletons is provided for all platforms (true portability requires the skeletons to be implementable equally efficiently on different machines).

Many have advocated the use of skeletons in parallel programming environments and in particular, research has focused on functional languages (Blelloch *et al*., 1993, Bratvold, 1993 and 1994; Danelutto *et al*., 1993; Darlington *et al*., 1993; Kuchen *et al*., 1994). This is mainly because skeletons can be expressed elegantly as higher order functions. These are a natural part of functional languages and have already been used widely in functional programming, not so much because they may embody parallelism, but because they provide a concise way of expression. Similar concepts form the basis of the idea to use the Bird-Meertens Formalism as a parallel programming model (Skillicorn, 1992).

On the other hand, little attention has been paid to the elementary weaknesses of skeletons: first of all, a set of skeletons has to be implemented on every platform and secondly, a given set may not be very suited to solve some problems efficiently or elegantly. It has already been pointed out by Cole that new skeletons will have to be developed and that 'the "ad hoc" implementation of each skeleton from scratch on each new architecture would result in much wasted effort' (although this effort is relatively little compared to that put into application development using crude tools). Considering this, it is not surprising that most experiments with some form of skeletons have been produced with a small set of data parallel languages on a limited set of machines that support these languages well (Blelloch *et al*., 1993; Cann, 1992). So far, we have gained only little experience with skeletons in more common functional languages on more general purpose machines (Bratvold, 1993, 1994; Kuchen *et al*., 1994). This indicates that an intermediate level of abstraction is desirable.

In this chapter, some examples will illustrate how Concurrent Clean can be used to implement a range of high level skeletons. In contrast to Darlington (1993), not only the meaning of each skeleton will be established by its functional language definition, but also its behaviour. Thus, we will use a lazy functional programming language with general mechanisms for unstructured parallelism only, to capture structured parallelism. This not only allows to build a structured system that can be extended easily, but it also permits mixing of structured and unstructured parallel programming.

Power of expression is important, but actual speed is crucial in a parallel system. Some experiments on the transputer system will make clear that no performance penalty needs to be paid, compared to parallel functional (Concurrent Clean) programs without skeletons (Bülk *et al*., 1993; Nöcker, 1993-c), and compared to other functional languages that provide skeletons in more restrictive ways (Bratvold, 1993; Kuchen *et al*., 1994). Some

Concurrent Clean programs are even competitive with C. But before we take a look at these examples, we will introduce some auxiliary functions.

8.2. Auxiliary functions

With the basic program annotations that Clean supports, it is possible to introduce and control parallelism at a very low level. Nonetheless, it turned out to be useful to add a few auxiliary functions. The implicit communication mechanism transports evaluated remote arguments to functions that need them, where they can be used for further processing. It is a little cumbersome to achieve the converse, that is: transport a function to a remote argument and apply it to this argument there. One would have to keep track of the location of an expression explicitly (see the definition of the buffering sieve of Erathostenes in the previous chapter). For this reason we have defined the following functions and types in Clean.

```
:: R x = Remote x

r_ap :: (x -> y) (R x) -> R y
r_ap f (Remote x) = r_ap_at (arg_id x) f x
where
    r_ap_at :: ProcId (x -> y) x -> R y
    r_ap_at processor f x = Remote ({Pn at processor} f x)

get_remote :: (R x) -> x
get_remote (Remote x) = x

id :: x -> x
id x = x
```

First of all, we have devised a polymorphic higher order function *r_ap* (remote application) that is able to transport a function to a remote argument and apply it to this argument. The result is again a remote object. It uses the low level function *arg_id* that returns the location of its argument; more precisely, it returns the position of the root node of its argument. This may seem to violate referential transparency, but, as processor ids only have a meaning within annotations, pureness is preserved in this 'para-functional' programming style (no operations have been defined on processor id's other than ***{Pn at ...}***). Even so, the *arg_id* function is completely hidden in the definition of the *r_ap* function. It is not available for programmers. In a sense, *r_ap* can be seen as a very basic skeleton that hides these confusing details.

In addition, we have introduced a new algebraic type that indicates - or rather suggests - that a data structure is at another processor. Typically, the argument of a *Remote* constructor will be on a different processor than the constructor itself (see figure 8.1). Such a type is useful for various reasons. Firstly, it helps the programmer to keep track of remote

structures. In Clean there is no obvious difference at the language level between a local object and a remote one. This is very convenient during default evaluation, for then we do not need to worry about communication. But if one wants to deal with certain remote objects in a special way, one would like some help in tracking these objects. This can be provided by declaring some objects to be remote. Secondly, the extra constructor *Remote* allows functions - always reducing to RNF - to return a truly remote result: a reference to a result at another processor. This becomes apparent in the definition of the *r_ap_at* function above. Without the extra constructor it would not only apply a function at a remote argument, but it would also subsequently evaluate (i.e. request) the result, so that it does not remain remote. This would not be the intended behaviour. And finally, having the *Remote* constructor, functions that have remote arguments need to perform a pattern match on the *Remote* constructor to get hold of the actual reference to the argument. This ensures that any indirections that may have been added after constructing the remote object will conveniently be removed automatically. This makes it easy for the *arg_id* function to find out the exact location of its argument. It will not accidentally return the location of some selector function for example.

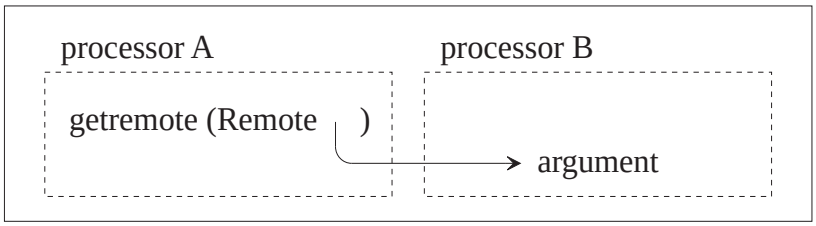


Figure 8-1: *The locations of the 'Remote' constructor and its argument in the expression 'get_remote (Remote argument)'.*

And finally, two simple functions have been defined. The *get_remote* function turns a remote object into a local one, using the standard evaluation and communication mechanism. A *put_remote* function is not very useful, as this can easily be provided by the standard ***{P^n}*** annotation in conjunction with the *Remote* constructor (see the definition of the *r_ap_at* function). The identity function is sometimes necessary to act as a copy-stopper. This is useful in case the *Remote* constructor is produced by a remote function, and not locally, as is the case in the *r_ap_at* function above. If so, we need to insert a copy-stopper between the constructor and the object, so that evaluation and transmission of the constructor does not trigger transmission of the object as well.

Combined with these functions, the standard Clean annotations are powerful enough to easily define a range of high level skeletons, as we will see in the following sections. We will construct skeletons that provide efficient data parallel constructs, skeletons to perform parallel I/O, and finally, skeletons that describe stream processing.

8.3. Skeletons for data parallelism

Unlike Sisal (Böhm *et al.*, 1989; Cann, 1989), Concurrent Clean is not a strict language dedicated to data parallelism. One of its major shortcomings has been the lack of suitable constructs for this kind of parallel processing. In this section we will show that functional

languages not only can be used as data parallel languages, but also to implement data parallel constructs efficiently.

8.3.1. Arrays and matrices

Arrays and matrices undoubtedly are the most prominent data structures in data parallel processing. Clean does have some support for matrices and arrays, but these are not 'data parallel' in the sense that they are automatically distributed over a number of processors. Instead, each array is associated with a single contiguous block of memory on a single processor, much like arrays in sequential imperative languages. In addition, referential transparency forces single-threaded access on these arrays if updates are to be done in place.

Despite this sequential nature of arrays in Clean, they are very suited for parallel programming. Even if referential transparency forces single threaded access on arrays, this can be better than using lists, which impose an order on accessing elements. And compared to lists, strict arrays can be considerably more efficient when they are to be transmitted to other processors. The previous chapter has shown that the use of arrays instead of lists can dramatically speed up parallel matrix multiplication, even if one uses a very straightforward algorithm without any data parallel constructs.

Below, we will demonstrate how an efficient implementation of data parallel matrices can be obtained from the standard matrices in Clean. This will allow efficient parallel processing on a large number of processors, in contrast to the examples in chapter 7, which were not very scalable. First we will choose the structure of the distributed matrix and define some - rather general - data parallel operations on it. Next, we will use these to construct a data parallel matrix multiplication program that is based on Gentleman's algorithm (1978). And finally, we will present the execution times of this program, which will make clear that building data parallel skeletons in Clean does not result in a performance penalty.

8.3.2. Operations on distributed matrices

The idea for obtaining a distributed matrix, is to define a local structure that refers to remote substructures, as depicted in figure 8-2.

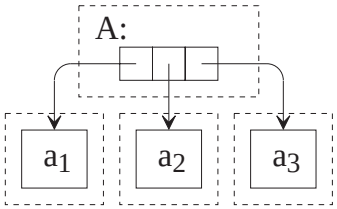


Figure 8-2: *A distributed array A, of which each part a_i resides on a different processor. The dotted lines indicate processor boundaries.*

Using the definitions given earlier we can simply define parallel matrices to be a double list of remote sub-matrices, as shown below. Choosing a list of lists is rather arbitrary though. One could have used a different structure, like for instance a quad-tree, or

a doubly linked circular list, or several, while using some natural transformation functions to get from one form to the other.

```
    :: PMatrix :== [[R Matrix]]
```

Map- and fold-like operations are crucial in data parallel programming languages. We will give Clean definitions of each kind below. These functions are polymorphic and operate on double lists of general remote objects. They can be applied to objects of type *PMatrix*, but also to different distributed objects that use a double list as local structure.

The *map3* function shown below constructs a new distributed structure out of three distributed structures. The remote elements of the new structure will contain *f a b c*, where *a*, *b* and *c* are the corresponding remote elements of the argument structures. The function *f* is applied at the processor that contains *c* and this will also be the location of the resulting element.

```
map3 :: [[R x1]] [[R x2]] [[R x3]] (x1 x2 x3 -> y) -> [[R y]]
map3 [row1 : rows1] [row2 : rows2] [row3 : rows3] f
    = [ map3row row1 row2 row3 f : map3 row1 rows2 rows3 f ]
map3 [] [] [] f = []

where
    map3row :: [R x1] [R x2] [R x3] (x1 x2 x3 -> y) -> [R y]
    map3row [e1 : e1s] [e2 : e2s] [e3 : e3s] f
        = [ r_ap (map3elem f e1 e2) e3 : map3row e1s e2s e3s f ]
    map3row [] [] [] f = []

    map3elem :: (x1 x2 x3 -> y) (R x1) (R x2) x3 -> y
    map3elem f e1 e2 e3 = f (get_remote e1) (get_remote e2) e3
```

The definition of *map3* is completely lazy: new processes will only be introduced at the moment the corresponding part of the resulting double list is needed by some other computation. Consequently, if one would pass the result of *map3* to some function that sequentially accesses the different parts of the result structure, no parallelism would be introduced, although different parts would be computed at different processors one after the other. This may seem odd, but our aim is to provide the kind of data parallel laziness that is proposed by Hill (1993). The parallel construct we have defined above is nothing more than a description of the *location* that certain computations will take place, should they turn out to be needed. This has the advantage that unnecessary computations will not take place. Our definition of the *map3* function could be even lazier still, because the *r_ap* function embodies a ***{P^n}*** annotation that actually *starts* a new process, while it would be sufficient to just *place* a function at the correct processor (lazy normal form copying will ensure that this function does not move to another location).

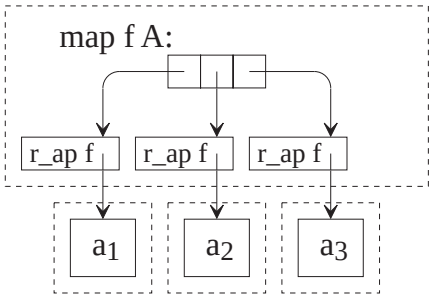


Figure 8-3: *Applying a lazy 'parallel' map to a distributed array A (see figure 8-2). The dotted lines indicate processor boundaries. No real parallelism is introduced here. However, as soon as some computation hits one of the r_ap nodes the unary function f will be applied to the corresponding argument at the remote processor.*

Having this lazy definition of map, an additional eager mechanism is required to actually introduce new processes where necessary. This is closely related to standard sequential lazy evaluation, where computations are driven by an eager printing mechanism. In our case true parallelism may be introduced by an eager *parallel fold* function, as the one presented below. It requests all remote elements of a doubly linked list in parallel.

```
fold :: [[R x]] (y y -> y) (x -> y) y -> y
fold [row : rows] h g e
    = fold rows h g ({I} foldrow row h g e)
fold [] h g e = e

where
    foldrow :: [R x] (y y -> y) (x -> y) y -> y
    foldrow [elem : elems] h g e
        = foldrow elems h g e'
        where
            e' = {I} get_remote (r_ap h' elem)
            h' = compose (h e) g
    foldrow [] h g e -> e

    compose :: (y -> z) (x -> y) x -> z
    compose h g x = h (g x)
```

The *fold* function above applies the function *g* to each remote element and folds the results with the function *h*. Normally, *g* will be some folding function over the structure of the remote elements. Most concurrency can be found in the parallel application of *g*, but *fold* is not as parallel as could be, because there is a sequential dependency between the applications of *h*. A trivial solution would be to split the double list recursively and fold it in a tree-like manner, provided that *h* is associative.

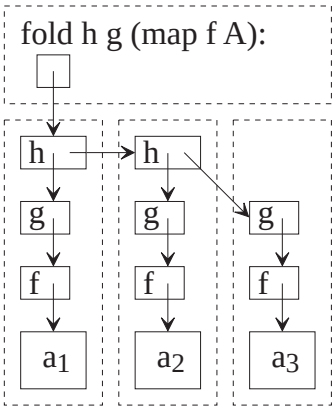


Figure 8-4: *The application of a parallel fold to the distributed array of figure 8-3. Again, the dotted lines indicate processor boundaries. After some reduction steps the situation above will arise. The function h will be applied in parallel on its arguments. This will force parallel evaluation of g, which in turn introduces the parallel evaluation of f (provided that h needs the result of g, and g needs the result of f). And finally, the result of the fold function will be copied to the root processor.*

Transformations like shifts and rotations can easily be defined on the local structure of the distributed object. They merely perform some permutations on the references to the remote structures. In this way, communications are automatically postponed until needed. This means that the transformation *rotate_left (rotate_up x)* does not introduce more communications than some function *rotate_left_and_up x*.

```
rotate_up :: [[x]] -> [[x]]
rotate_up [row : rows] = append rows [row]
rotate_up [] = []
```

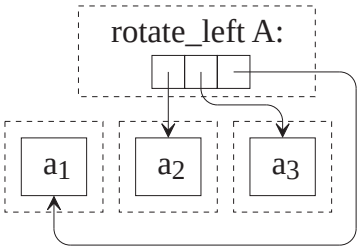


Figure 8-5: *Rotating the distributed array A (see figure 8-2) to the left. This merely involves a pointer permutation. No communication takes place.*

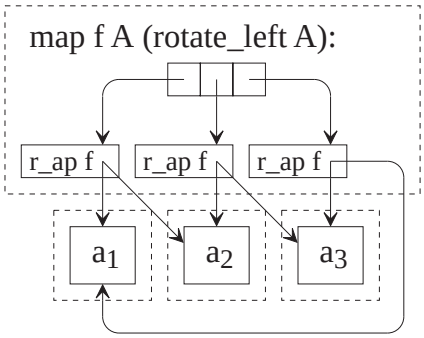


Figure 8-6: *Mapping a dyadic function f on A and on (rotate_left A). Communication will only take place if f is actually executed at the location of one of its arguments. Depending on the definition of the map function f will either be executed at its first argument or at the second. If f is applied at the location of its first argument the elements of A will actually be rotated to the left. Otherwise, they will be rotated to the right. The overall effect remains the same.*

Broadcasts

Broadcast communication can be provided by means of distributed objects as well. For example, it is possible to define a primitive function *broadcast* that has the following type:

```
broadcast :: x -> [R x]
```

The effect of this function will be that it broadcasts its argument to all processors, and delivers a list of remote objects. Element e_i will refer to the copy of the original argument at processor *i*. Having obtained such a broadcast object, we can simply start up functions at the appropriate processors using the *r_ap* function. Alternatively, we might also distribute the list of remote objects over all processors (only copying references, *not* the remote elements themselves), so that each processor obtains a complete overview of the location of each copy. Again, we might use the broadcast function to accomplish this:

```
broadcast (broadcast x) :: [R [R x]]
```

Supporting such a double broadcast operation requires careful reconsideration of the weighted reference counting scheme. In large networks one introduces many references to the same object, copying the same channel node many times, each time halving its weight. Consequently a straightforward solution could introduce many indirection nodes, in particular in large networks.

And concluding, distributed objects can be created with functions like the map function above. Only now, each occurrence of *r_ap f x* should be replaced by *Remote* ***{P^n at …}*** *(create_element …)*. Again, the *create_element* function can be passed as an argument.

Note that the map and fold operations defined above just happen to be specialised to lists of lists. Constructing these operations for other kind of data structures is fairly straightforward. It comes down to walking the particular structure while distributing and inserting functions in a similar way. It will be interesting to exploit type classes and constructor classes and define skeletons for sets of data structures.

8.3.3. Data parallel matrix multiplication

The functions that we have defined in the previous paragraphs allowed us to implement Gentleman's matrix multiplication algorithm (Gentleman, 1978) in a straightforward way. We used a few additional functions for rotating and skewing matrices, but these can be constructed similarly to the *rotate_up* function above. Gentleman's algorithm lets each processor compute a part of the resulting matrix, by rotating the relevant rows and columns 'through' this processor (see also figure 8-7). It does this for each part of the result simultaneously. Below, we have listed the main part of the program.

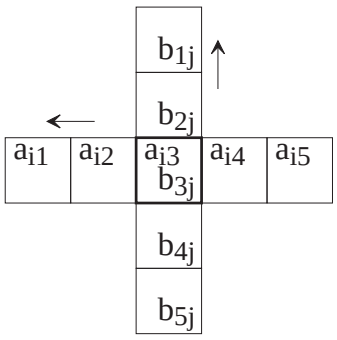


Figure 8-7: *A snapshot of Gentleman's matrix multiplication algorithm. The bold square depicts the processor that will multiply row a_i with column b_j. It has already computed $a_{i1} \times b_{1j} + a_{i2} \times b_{2j}$, and it is currently computing $a_{i3} \times b_{3j}$. The elements a_{i1}, a_{i2}, b_{1j}, and b_{2j} have been passed on to other processors. The remaining elements still have to pass through this processor. At the same time the other processors will be working on other parts of the result in a similar way.*

The *new_mat* function creates a new distributed matrix and fills it according to its argument function. The skew and rotate functions have been defined in the same way as the *rotate_up* function above. *length* is a function that computes the length of a list. It tells how many rotations are necessary. *mul_add* does all the work. It is a sequential function that multiplies two ordinary matrices and adds the result to another (this is very similar to the matrix multiplication algorithms in the previous chapter).

```
mul :: PMatrix PMatrix -> PMatrix
mul a b = mul' a' b' zero rotate_n
where
    a'       = skew_left a
    b'       = skew_up b
    zero     = new_mat zero_mat
    rotate_n = length a

    mul' :: PMatrix PMatrix PMatrix Int -> PMatrix
    mul' a b c 0 = c;
    mul' a b c n
        = mul' a' b' (map3 a b c mul_add) (n - 1)
        where
            a' = rotate_left a
            b' = rotate_up b
```

The whole computation is driven by the function *fold* as shown below. This is not part of Gentleman's algorithm, but it is merely used to guarantee parallel evaluation. If we omit the fold function the whole resulting matrix gets printed on screen. This happens lazily, so we loose parallelism, while gaining I/O overheads. Conversely, if we merely select a single sub-matrix from the result, only this element will be computed on its own processor. The rest will remain idle. The fold function avoids this. It ensures that all elements of the result are computed in parallel.

```
fold (mul a b) (&&) matrix_to_bool True
where
    a = new_mat some_mat
    b = new_mat another_mat
```

8.3.4. Performance Measurements

The following tables present the results that have been obtained with matrices containing floating point numbers of 64 bits. The parallel versions have been compared to a purely sequential matrix multiplication program that contains no overheads for parallelism (using 2-dimensional arrays, see also table 7-3). Although the actual network topology matches the matrix structure, no effort has been made to place sub-matrices that are neighbours logically at physically neighbouring processors (for our hardware provides no clear relation between processor numbering and the physical network). This increases communication overheads, especially for large networks and small matrices.

Table 8-1: *Execution times of matrix multiplication in Clean*

	sequential	4 processors	16 processors	64 processors
multiply 32 × 32	1.0 sec.	0.4 sec.	0.4 sec.	1.7 sec.
multiply 64 × 64	7.6 sec.	2.1 sec.	1.0 sec.	1.8 sec.
multiply 128 × 128	60.3 sec.	15.6 sec.	5.1 sec.	2.9 sec.
multiply 256 × 256	480.7 sec.	121 sec.	36.3 sec.	11.3 sec.
multiply 512 × 512	3846 sec.*	966 sec.*	281 sec.	73.8 sec.

Table 8-2: *Speed-up of matrix multiplication in Clean*

	4 processors	16 processors	64 processors
multiply 32 × 32	× 2.5	× 2.5	× 0.6
multiply 64 × 64	× 3.6	× 7.6	× 4.2
multiply 128 × 128	× 3.9	× 11.8	× 20.8
multiply 256 × 256	× 4.0	× 13.3	× 42.5
multiply 512 × 512	× 4.0*	× 13.7*	× 52.1*

The star indicates where we have used estimations, as matrices of size 512 by 512 are too big to be accommodated on a single processor. Matrix multiplication is of $O(n^3)$, so it is not unreasonable to assume it will take 8 times longer to multiply two matrices of size 512 by 512 on one processor, than it takes to multiply two of size 256 by 256. This assumption is strongly supported by the results we have obtained for the sequential program.

Table 8-3: *Execution time and speed-up in Clean after inlining the array selection functions inside the innermost vector multiplication loop.*

	sequential	16 processors	speed-up	64 processors	speed-up
multiply 32 × 32	0.3 sec.	0.4 sec.	× 0.8	1.7 sec.	× 0.2
multiply 64 × 64	2.3 sec.	0.6 sec.	× 3.7	1.8 sec.	× 1.3
multiply 128 × 128	17.6 sec.	1.8 sec.	× 9.6	2.5 sec.	× 7.0
multiply 256 × 256	139 sec.	10.2 sec.	× 13.6	5.0 sec.	× 27.8
multiply 512 × 512	1112 sec.*	73.9 sec.	× 15.1*	22.1 sec.	× 50.3*

As in the previous chapter, the absolute performance can be improved by inlining some array selection functions. table 8-3 lists the resulting performance figures (these are comparable to C, see also table 7-5). Using parallel matrices has two considerable advantages over the solutions of chapter 7. First of all, larger matrices can be used, as they do not have to fit in the memory of a single processor. And secondly, this algorithm is far more scalable. This is already noticeable for 16 processors (compare table 8-3 with table 7-

6 and table 7-7). With 64 processors we get significant speed-ups, whereas earlier versions became slower if more than 16 processors were used.

The figures presented above compare favourably to the ones presented for π-RED+ (Bülk *et al.*, 1993) and the Data Parallel Functional Language (DPFL) presented in (Kuchen *et al.*, 1994). Both give similar speed-ups - albeit with smaller networks -, but they have worse absolute performance (No absolute performance figures have been presented for π-RED+, but this is an interpreter). Other implementations of functional languages on machines with distributed memory either do not list results for matrix multiplication, or they are considerably less efficient.

Table 8-4: *Execution times of Clean and DPFL for multiplication of matrices of size 500 × 500. For Clean we did not use inlining of array selections this time.*

	4 processors	25 processors
Concurrent Clean (25 MHz T800)	902 sec.*	169 sec.
DPFL (20 MHz T800)	3127 sec.	510 sec.

More interesting than execution times however, - as this largely depends on sequential code quality -, is that DPFL uses a set of custom data parallel constructs. To implement these, a communication mechanism has been suggested that uses version numbers to maintain the correct ordering of messages and operations. Though messages may pass each other in Clean as well, we do not need such a construct, because the correct order of operations is automatically maintained by data dependencies. In addition, we do not rely on uniqueness properties as heavily as DPFL. Instead of requiring each instance of an entire array to be unique, we only need this property temporarily during sequential construction of some remote sub-array. Usually, this requirement will be met by the array-constructing argument functions of the skeletons. We have not (yet) made use of uniqueness properties within the definition of the skeletons themselves.

We have not been able to compare Clean with SISAL in a sensible way. To some extent because Clean does not optimise loops on arrays, but mainly because the architectures for which they are available differ greatly. The performance figures obtained on transputers are not conclusive with respect to performance on a Cray (and vice versa). We end up comparing architectures instead of languages. Eventually, an important question will be whether laziness allows the same level of code optimisation (copy elimination) as has been achieved in SISAL.

8.4. Skeletons for parallel I/O

I/O can be a bottleneck if only one process is allowed to do it. Typically, machines have many independent devices for I/O that all operate in parallel. A single thread of control may not be able to drive all of these efficiently, either because it is too slow, or because it introduces unnecessary sequentialisation of I/O. The latter is not unlikely in a pure functional language without a non-deterministic merge function. Instead, it would be best to split up the load and let different processes - and processors - perform various I/O tasks in parallel.

To accomplish this, one needs to divide an I/O system at some level into several independent parts. For instance, one could view a picture as a set of independent pixels. Different processes may then colour these pixels in parallel, which is all right as long as the order in which this happens does not matter. If we are only interested in the final result, which will always be the same, this is no dangerous form of non-determinism.

At some level however, this concept is unmanageable. There are moments one would rather like to manipulate a picture as a whole, or handle a properly ordered sequence of pictures, or deal with some other object in which the pixels are united, without loosing the efficiency of parallelism. In these cases one needs to supply parallel I/O within some framework. This ordered concurrency can be provided by skeletons in exactly the same way as they provided data parallelism. Only now, we are not working with parallel data structures, but with parallel I/O devices.

8.4.1. Plotting pixels in parallel

We will stay with the picture example and show how to use the two skeletons below to build and draw a picture in parallel. Again, these skeletons have been constructed in Clean with the primitives presented above.

```
new_PPict :: Size (Int Int -> Pixel) -> PPict
plot_PPict :: Window PPict -> Window
```

The *new_PPict* function is almost the same as the data parallel *new_mat* function. It lazily creates a distributed picture, in which each remote part is a little square of pixels that represents a distinct part of the image. The only difference with the matrix creating skeleton is that it does not build a local list of lists, but instead it constructs a distributed one. We will see below why this is useful. *new_PPict* has two arguments. The first is a tuple of four integers that describe the size of the picture (respectively, the horizontal and vertical size of the picture and the horizontal and vertical size of each remote square). The second argument is a function that describes the scene. It delivers the colour of each pixel according to its co-ordinates in the image.

plot_PPict is able to draw such a distributed picture in a window, and it does this for all image parts in parallel. It has been defined analogous to the *fold* function and drives parallel computation by eagerly applying a plotting function to each remote part of the image simultaneously (similar to *g* in *fold*). For this it not only splits the picture recursively into its basic components, but it also breaks down the window structure into corresponding sets of pixels. The result of the various plotting functions is then combined to form a new window, similarly to the function *f* in *fold*. In this way windows can be treated as a whole, while incorporating parallelism.

We have tested these functions in two programs, of which the main calls have been listed below. The execution times can be found below. In both cases, the *fill* function passed to *new_PPict* is a plain sequential function that does not contain any construct for parallelism. Conversely, *new_PPict* itself scatters a considerable number of processes over the network: 448 and 900 respectively. One program is quite simple and draws a man-

delbrot set. The other is fairly extensive and generates a scene of polygons using a ray-tracing algorithm. To accomplish this, it needs to distribute a collection of objects - the 'scene' - over all processors. This is done automatically by the *r_ap* function contained in *new_PPict*, but to avoid doing it from a single processor for every process, we have kept the *PPict* structure as distributed as possible: it is structured as a distributed tree (using broadcasts may have been better, but the graph copying mechanism does not support this yet).

```
plot_PPict Window (new_PPict size fill)
where
    size = (560 320 20 20)
    fill = mandelbrot_color

mandelbrot_color :: Int Int -> Int
mandelbrot_color x y = ...
```

```
plot_PPict Window (new_PPict size fill)
where
    size = (300 300 10 10)
    fill = trace_colour screen scene

traceColour :: Screen Scene Int Int -> Int
traceColour screen:(h,v,origin,eye) scene x y = impactColour impact
    impact = firstImpact ray scene
    ray    = (eye, vNormalise (vSubtract point eye))
    point  = vAdd origin (vAdd dv dh)
    dh     = vMultiply h (x / detail)
    dv     = vMultiply v (y / detail)
    detail = 300

firstImpact :: Ray Scene -> Impact
firstImpact ray [object | rest]
  = closest (testForImpact ray object) (firstImpact ray rest)
firstImpact ray [] = NoImpact

...
```

Table 8-5: *Execution time with parallel plotting*

	1 processor	16 processors	32 processors	64 processors
mandelbrot	153.5 sec.	13.6 sec.	9.2 sec.	6.3 sec.
raytrace	331.3 sec.	33.0 sec.	17.4 sec.	9.9 sec.

Table 8-6: *Speed-up with parallel plotting*

	16 processors	32 processors	64 processors
mandelbrot	× 11.3	× 16.7	× 24.4
raytrace	× 10.0	× 19.0	× 33.5

Although it seems that first a picture is generated, and then plotted, this is not the case. As in the data-parallel map, construction of the image is lazy, and therefore will take place at the moment the plotting function requires it: when it starts to plot a basic part of the picture. So reading ';' as 'followed by' and || as 'in parallel', instead of $((gen_1 \,||\, gen_2 \,||\, ... \,||\, gen_n)\ ;\ (plot_1 \,||\, plot_2 \,||\, ... \,||\, plot_n))$, we get $((gen_1;\ plot_1) \,||\, (gen_2;plot_2) \,||\, ... \,||\, (gen_n;plot_n))$. This shows better on screen what is going on, while avoiding unnecessary sequentialisation: each part of the image is plotted as soon as it is available. This spreads plotting over time, which is more efficient in many cases.

Table 8-7: *Execution time with sequential plotting*

	1 processor	16 processors	32 processors	64 processors
mandelbrot	155.5 sec.	18.5 sec.	12.7 sec.	9.2 sec.
raytrace	392.5 sec.	36.1 sec.	22.4 sec.	16.8 sec.

Table 8-8: *Execution time without plotting*

	1 processor	16 processors	32 processors	64 processors
mandelbrot	141.5 sec.	13.6 sec.	9.2 sec.	6.3 sec.
raytrace	321.0 sec.	31.3 sec.	16.7 sec.	9.9 sec.

In our system parallel plotting was able to improve execution time considerably - up to a factor of 1.7 -, compared to having a single process that draws the image sequentially (see table 8.7). The latter causes all pixels to be plotted in a fixed order, as in $(gen_1 \,||\, ... \,||\, gen_n \,||\, (plot_1;\ ...\ ;plot_n))$. As a result, most of them are plotted in a single burst after computing the whole picture. This increases execution time notably because the underlying system transfers all pixels from the transputer network to a server via a single transputer link (which also makes it impossible to test truly parallel I/O). Using parallel plotting, transferring the pixels gets overlapped with ordinary computation to such an extent that plotting overheads are barely noticeable (compare table 8.5 and 8.8).

The mandelbrot program in Clean is about a factor 1.5 slower than an iterative version in C. This is caused by stack management overheads that are not present in the iterative C version. Concurrent Clean additionally suffers from having to manage multiple stacks on the transputer (see also table 4-5 and table 4-9). Recursive versions in C on the other hand, perform worse than Clean, even after some tuning. Compared to SkelML (Bratvold, 1993), which is strict, the ray tracer in Clean is a factor 3.5 faster, although the speed-ups are comparable (see also table 5-4; SkelML uses Occam as an intermediate

language, whereas the Clean system generates transputer assembly directly). As with the matrix multiplication example, we have not been able to compare our results in a sensible way with other implementations of functional languages on distributed memory machines.

Load imbalance - caused by the irregular structure of these problems - largely explains the limited speed-up. The use of skeletons does not degrade performance here. Generating a distributed pixel map is not more difficult than generating a distributed matrix. This means that speed-ups can be at least as good as in table 8-2 and table 8-3 if we create an image by filling it with a function that takes constant time for all co-ordinates.

8.5. Skeletons for streams

Our last example involves the use of skeletons to construct streams. These provide pipelines and process networks (which can be used to simulate systolic arrays, for example). To some extent, the low level constructs for parallelism in Clean are very well suited to express this kind of parallelism directly. This also holds for certain divide-and-conquer style programs, which means that we may not always need to employ skeletons. Examples are well-known programs like ‘queens’ and the sieve of Erathostenes. Using basic Clean constructs, one can obtain good speed-ups for both in a rather straightforward way, although the sieve requires a special buffering function between distinct filter functions. These examples are very simple however, and inserting special functions, such as buffers, may be hard sometimes. In these cases, skeletons are invaluable.

Before we proceed, we will present a modified version of the *buffer* function that was introduced in chapter 7. It can be defined in a more concise way with the functions we have introduced in section 8.2. Below, we have listed the buffer functions that have changed. Note that one does not have to pass a processor id to the buffer function anymore. It will automatically start up the *write* function at the right processor.

```
:: Buffers = (!Array, R Buffers)

buffer :: Int (R [Int]) -> [Int]
buffer size list = read (get_remote (r_ap (write size) list))

write :: Int [Int] -> Buffers
write size list = fillbuffer (createbuffer size) size list
where
   fillbuffer buf size list = (filledbuffer, (Remote nextbuffer))
   ...

read :: Buffers -> [Int]
read (filledbuffer, nextbuffer)
    = [first : get rest {I} (get_remote nextbuffer)]
    ...
```

As a result, it becomes possible to express the buffered sieve algorithm of the previous chapter in a less awkward way, without using the *currentP* function.

```
sieve n [p : s]
    | n == 0    = [p : {P^n} psieve ns (Remote f)]
    | otherwise = [p : {I} sieve (n - 1) f]
    where
        f  = filter s p
        ns = squareroot p
sieve n [] = []

psieve n list = sieve n (buffer BufferSize list)
```

Using this buffering function, we can create complex skeletons for efficient stream processing. Below, we have listed one that is able to construct a tree of buffered stream processes. It gets a tree of remote lists as an argument and two functions: *node_f* and *leaf_f*. The function *leaf_f* is applied to each list in the tree in parallel (at the leaves), which results in horizontal parallelism. The *node_f* function is placed at each node and combines the results. This leads to several vertical streams that flow from the leaves down to the root of the tree. To regulate the flow, buffering functions are inserted where appropriate, that is, between functions executing on different processors. The size of the buffer gets halved each step up the tree. Note the similarity with the fold and map functions defined earlier.

```
buffertree :: (Tree(R[x])) ([x][x]->[x]) ([x]->[x]) -> [x]
buffertree list node_f leaf_f
    = buffer BuffSize (Remote {P^n at ItoP StartProc} root)
    where root = buffertree' list node_f leaf_f StartProc BuffSize

where
 buffertree' :: (Tree(R[x])) ([x][x]->[x]) ([x]->[x]) Int Int -> [x]
 buffertree' (Leaf list) node_f leaf_f proc size
   = leaf_f (buffer size list)
 buffertree' (Node left right) node_f leaf_f proc size
   = node_f arg1 arg2
   where
     arg1 = buffer newsize (Remote {P^n at ItoP leftP} buffer_left)
     arg2 = buffer newsize (Remote {P^n at ItoP rightP} buffer_right)
     buffer_left  = buffertree' left node_f leaf_f leftP newsize
     buffer_right = buffertree' right node_f leaf_f rightP newsize
     leftP  = newleftP proc
     rightP = newrightP proc
     newsize = Half size
```

This skeleton can be employed to easily implement the following divide-and-conquer style merge-sort algorithm. It sorts a tree of unsorted lists by applying a sequential sorting function at each leaf and executing a merge function at every node. The stream-like processing overlaps these computations as much as possible.

```
mergesort :: (Tree (R [Int])) -> [Int]
mergesort list = buffertree list merge sort

sort :: [Int] -> [Int]
sort [x]   = [x]
sort list  = merge (sort left) (sort right)
where (left, right) = split list [] []

merge :: [Int] [Int] -> [Int]
merge alist=:[a : as] blist=:[b : bs]
    | less a b   = [a : merge as blist]
    | otherwise  = [b : merge alist bs]
merge alist []   = alist
merge [] blist   = blist

split :: [Int] [Int] [Int] -> ([Int],[Int])
split [a : [b : rest]] left right
    = split rest [a : left] [b : right]
split [a] left right = ([a : left],right)
split [] left right  = (left,right)
```

Table 8-9 lists the results of this program for different input sizes. The tree of unsorted lists is generated at a single processor. Two versions have been tested. One that uses the standard integer comparison, and one that employs a complex comparison. Compared to earlier experiments with special stream functions in Clean (Nöcker, 1993-c), the use of buffers gives better performance, while also improving memory usage.

Table 8-9: *Execution time for parallel mergesort.*

	1 processor	8 processors	16 processors	32 processors
standard 20000	15.6 sec.	4.6 sec.	3.5 sec.	3.9 sec.
standard 40000	59.8 sec.	11.2 sec.	8.0 sec.	9.1 sec.
complex 20000	79.2 sec.	19.4 sec.	12.2 sec.	10.9 sec.
complex 40000	196.0 sec.	42.6 sec.	25.3 sec.	21.7 sec.

Table 8-10 *Speed-up for parallel mergesort.*

	8 processors	16 processors	32 processors
standard 20000	× 3.4	× 4.5	× 4.0
standard 40000	× 5.3	× 7.5	× 6.6
complex 20000	× 4.1	× 6.5	× 7.3
complex 40000	× 4.6	× 7.8	× 9.0

The reason that speed-ups are limited is two-fold. First of all, - despite the use of buffering functions -, delays can be substantial. Our buffering mechanism is too simple to avoid all delays, which may be caused by complex data dependencies. And secondly, the algorithm contains only a limited amount of inherent parallelism. About half of the available processors is used to sort the remote lists in parallel and the rest for merging the results. One cannot be faster than the time it takes to sort one sub-list, which is a problem for small networks. Conversely for large networks, having a tree of processes, the root becomes a bottleneck. At a certain point, adding more processors does not help. It will even harm, as it causes more overhead at some processors (for example, at the list generating processor), while also introducing longer data paths. The effects can be quite complex and account for the speed-down for 32 processors compared to 16 with the standard integer comparison.

Having poor speed-ups for this particular program does not mean that stream-like processing is not useful at all. First of all, better speed-ups can be expected for programs that have more regular data dependencies, such as systolic arrays of the form $f_1\ (f_2\ (...\ f_n))$, where n is sufficiently large. Even the sieve of Erathostenes gives better speed-ups - up to 16 on 32 processors -, although the first filtering process forms a bottleneck (note however that it is not very useful to try to fit the sieve program into some skeleton structure: this kind of processing can very well be expressed with the primitives for parallelism). Secondly, many large computations contain compositions of functions working on streams. These computations can be overlapped to some extent. There may not be enough overlap to keep many processors busy, but the speed-up may be high enough nonetheless.

The importance of this becomes more apparent if we consider that data-parallel operations can easily be combined with stream-like processing. The latter may be used for consuming small numbers of processors, while the former is responsible for the big speed-up factors. For instance, one could map a stream-like parallel signal processing function - consisting of several filters - over a distributed database of signals. Each signal processing function will account for only a limited speed-up, but it does effect a better utilisation of processing power. Skeletons as defined above are powerful tools to accomplish this. Being higher order functions, skeletons for stream-like processing can be passed to skeletons for data parallelism (and vice versa), so that a new skeleton is composed out of old ones.

```
map_filters :: (Struct s) [s -> s] -> Struct s
map_filters database filters
  = map database (compose_stream filters)
```

8.6. Conclusions

Using Concurrent Clean we constructed a number of skeletons. In this way we easily obtained efficient skeletons for data parallelism, for parallel I/O, and for streams. They have been tested and show very good performance compared to more restrictive solutions. Constructing skeletons in this way makes it easy to extend a system with new skeletons, tailored for specific needs.

The primitives presented above allow functions to ‘travel’ over distributed structures easily. This offers an interesting starting point for research on distributed functional databases, using skeletons for searching and updating. These skeletons may then be parametrised with functions that perform local transformations on the database.

Conclusions

We started this thesis with an explanation of the techniques that we employed to realise an implementation of Concurrent Clean on transputer hardware. It became apparent that a large number of technical and fundamental problems need to be addressed in order to support the ***{P}*** and ***{I}*** annotations to their full extent. The main implementation topics were: efficient and reliable communications, the logical structure of the implementation, code generation, graph copying and garbage collection. The decision to make no concessions with respect to the generality of the implementation greatly influenced our design decisions. Although increasing the implementation work, this generality did not result in serious runtime overheads.

Reconsidering the research questions of chapter 1, we see that the ***{P}*** and ***{I}*** annotations are very general. We have been able to base a fair number of parallel algorithms on these simple annotations. Two problems became apparent however. First of all, it turned out that the original graph copying strategy did not provide clear runtime semantics: sometimes annotations were needed in awkward ways in order to get the desired behaviour. The new lazy normal form copying strategy - introducing the $\{P^n\}$ annotation - has solved this. And secondly, the $\{P^n\}$ and ***{I}*** annotations alone do not suffice to program concise solutions. As we have shown in chapter 8, it is possible to solve this problem if one uses the basic annotations to construct skeletons for parallelism. Thus, we obtained skeletons that are adequate for programming comprehensible parallel programs. A considerable advantage of defining skeletons in a functional language, is that new (and very specialised) skeletons can easily be added. Amongst others, we heavily depend on the higher-order features of Concurrent Clean.

With respect to performance, we have demonstrated that a number of parallel programs run very efficiently on our implementation. This not only holds for the programs that directly use $\{P^n\}$ and ***{I}*** annotations, but also for the ones that employ skeletons. However, arriving at an efficient solution often requires careful consideration of the data structures that are employed. As we have seen in chapter 7 the use of arrays (i.e. flat data structures) can be crucial for good parallel performance.

In order to allow flat data structures to be used efficiently in Concurrent Clean, one needs uniqueness typing. Unfortunately, as we have shown in chapter 6, uniqueness typing is incompatible with the original lazy graph copying strategy. The introduction of the lazy normal form copying strategy has solved this problem as well, as it is safe with respect to uniqueness typing.

Reasoning about parallel performance

Our implementation efforts have specifically been directed at machines with distributed memory. The reason for this is, that machines with distributed memory are more scalable than shared memory machines. The most important disadvantage of distributed memory machines is that they are difficult to program.

Our implementation aimed at reducing the complexity of programming distributed memory machines. This has only partly succeeded. On the one hand, it became possible to program such machines without having to use explicit message passing. This makes it easy to reason about the correctness of a program. On the other hand, reasoning about the efficiency remains difficult, although the introduction of normal form copying has slightly improved this situation. Let us consider the costs of a very basic parallel program.

```
merge ({Pn} f a) ({Pn} g b)
```

The meaning of such a program is that *f* and *g* are computed in parallel at different processors. The arguments *a* and *b* are evaluated locally, as soon as *f* and *g* need them. And finally the results of *f* and *g* are returned as soon as the *merge* function needs them. Suppose that the affix *t* stands for the costs of transmitting a result and *c* for the costs of computing it. The costs of exporting work - denoted by w_t - will be assumed to be constant. Furthermore, let the affix *p* and *u* denote the costs of packing and unpacking a message. The total costs can now roughly be approximated by the following formula (although the communication costs are not totally independent).

$$a_c+a_p+b_c+b_p+w_t+maximum(a_t+a_u+f_c+f_p+f_t\,,\, b_t+b_u+g_c+g_p+g_t)+f_u+g_u+\ merge_c$$

Often, *a*, *b*, *f* and *g* are needed computations, and in addition, they sometimes represent complex structures. If this is the case, evaluating them in a lazy manner will introduce unnecessary overheads. Not only because the computational overheads for lazy evaluation are often higher than for eager evaluation, but also because lazy transmission may result in many small messages and large delays. To deal with this, the compiler and the programmer can introduce eager evaluations on strict arguments. If one eagerly reduces *a*, *b*, *f* and *g* to normal form before transmitting the result, this evaluation strategy is equivalent to the sandwich evaluation mechanism of Wybert (Langendoen, 1993). The performance approximation basically remains the same, but often with considerably smaller figures for the transmission costs.

Another decrease in costs can be obtained by using data structures for which the packing and unpacking costs are small. Some structures, such as strict arrays, do not need any (un)packing at all. Avoiding these costs results in the following formula.

$$a_c+b_c+w_t+maximum(a_t+f_c+f_t\,,\, b_t+g_c+g_t)+\ merge_c$$

The main problem with the strategies above, is that they require f_C and g_C to be fairly large compared to the other costs. Otherwise, parallel evaluation will not be much better - or even worse - than sequential evaluation (which costs $a_c+b_c+f_c+g_c+merge_c$). In particular, this is a problem for programs running on distributed memory machines, because the costs of transportation are relatively high on such machines. Concurrent Clean allows the programmer to deal with this by introducing pipelined computations (using the ***{I}*** annotation to drive the pipelines). In this way, computations and communications can be overlapped. Ideally, the computations overlap totally and the transmission costs vanish, except for the initial costs of exporting work and the basic costs to set up a pipeline ($pipe_c$). This results in the following formula.

$$2 \times pipe_c+w_t+maximum\ (a_c+a_p+b_c+b_p+f_u+g_u+merge_c,\ a_u+f_c+f_p,\ b_u+g_c+g_p)$$

In reality, it will be extremely hard to realise such performance. Programs will need to be tuned to balance computation and communication. Buffering techniques are necessary to avoid delays. In some cases, it will be more important to avoid delays than to avoid unnecessary computations. If so, some computations will have to be computed speculatively. In short: it will take considerable efforts to obtain a highly efficient pipelined program.

To conclude, we stress the importance of choosing the right placement of processes and data. For example, the primitive Concurrent Clean annotations place functions at a particular location and then data is transported to these locations for processing. However, some algorithms perform better if data is placed at a particular location and functions are transported to the data and executed there. Gentleman's matrix multiplication algorithm uses this form of processing (see chapter 8). It avoids the costs of transporting the arguments (a_p, a_t, a_u, b_p, b_t, *and* b_u). In addition, the arguments (a_c and b_c) are computed in a distributed manner. The computational costs are approximately as follows.

$$w_t +maximum(a_c+ f_c+f_p+f_t\ ,\ b_c+g_c+g_p+g_t)+f_u+g_u+merge_c$$

The design of efficient parallel algorithms

As we argued in the first chapter, functional languages are inherently parallel, and the differences between sequential and parallel algorithms should be minimal (modulo annotations). Indeed, annotations cannot compromise correctness, but unfortunately, achieving good performance by merely placing annotations is not easy, and sometimes even impossible. Functional programs *are* inherently parallel, but they can still incorporate too much unnecessary sequential data dependencies. In some cases it will be necessary to avoid certain forms of sequentialisation and derive an algorithm that has a completely different structure. For example, if we compare the matrix multiplication results of chapter 7 and 8, we see two considerable differences. First of all, the structure of the algorithms differs notably, and secondly, the performance figures of Gentleman's algorithm are much better for large matrices.

This makes clear that *if* one starts with a sequential program, it will be necessary to develop tools for helping the programmer to transform a given sequential solution to an efficient parallel one. Some form of semi-automatic transformational reasoning will be required. However, realising such techniques is far from trivial, and one may question whether the strategy of parallelising sequential programs will be fruitful.

Parallel programs are not always difficult to reason about. For example, Gentleman's data parallel matrix multiplication algorithm is not a sequential algorithm: it would not make much sense to run such an algorithm on a single machine. In contrast, it is far more parallel than the divide-and-conquer solutions of chapter 7. Reasoning about Gentleman's algorithm implies reasoning about many concurrent processes at once. One does not think about Gentleman's algorithm in a divide-and-conquer manner where the results of a few argument processes are recursively combined. Instead, one envisages a considerable number of processes, all interacting collectively. Still, this does not complicate reasoning about the algorithm: Gentleman's solution is elegant and clear.

This shows that it will not always be wise to start programming a correct sequential solution and to transform in into a parallel one. It is often better to devise a parallel algorithm from the start and program it directly in a functional language.

To maximise parallelism, one should consider the development of techniques that help postponing potentially limiting design decisions such as choosing the right data structures. Data structures introduce certain forms of sequentialisation and they influence communication costs, as we have seen in chapter 7. If we have a look at the matrix multiplication algorithms, it is clear that the solution actually only has to be specified in terms of relations between the input and the output. However, at some point in time during programming we *have* to choose some suitable structure for combining the data elements, and this in turn, introduces an order on the operations that can be performed on these data elements.

List comprehensions - and skeletons in general -, are a good way to delay such decisions. If a solution is based on skeletons, it is possible to change the underlying structure of a program radically, by changing the implementation of the skeletons. On the other hand, by choosing some skeleton one also introduces a particular structure of processing. This may be too limiting as well. Perhaps one needs a programming paradigm that more radically separates the logical structure of a program from efficiency issues.

Future Work

Clearly, much is still unknown with respect to the implementation of functional languages. First of all, we still do not have much experience with large parallel applications written in a functional languages like Concurrent Clean. Consequently, it remains to be seen how well our implementation performs on such programs. In particular, it will be interesting to know what the effects of lazy normal form copying would be on writing large programs.

Secondly, load imbalance causes bad performance in certain cases. Compile-time techniques will not be sufficient general, and so, a runtime mechanism should be developed that deals with this problem in a convincing way. It may be necessary to move running processes from heavily loaded processors to lightly loaded ones.

Thirdly, not all garbage collection problems have been solved. In particular, one will have to tackle unbridled speculative parallelism and distributed cyclic structures. In this thesis we have shown a direction that might be taken to solve this problem, but it remains to be seen whether this approach is fruitful.

Fourthly, there is a growing base of heterogeneous loosely coupled networks that incorporate a vast number of processors (e.g. internet). Consequently, there will be need to dynamically manage code for different platforms, and reduce communication overheads as much as possible. For such networks it becomes worthwhile to reconsider compression techniques for graphs. Related to this, will be the growing significance of modelling a distributed I/O system.

And finally, one will need to have better tools for explicitly devising efficient (parallel) functional programs, considering the differences in speed that are caused by different algorithms and data structures. The support for reasoning about the *correctness* of functional programs is unparalleled, but the lack of techniques for reasoning about their *efficiency* is still one of their major shortcomings.

Bibliography

Achten **(1995)** P.M., and Plasmeijer M.J. 'The ins and outs of Clean I/O'. In *Journal of Functional Programming*, **5**(1), January 1995, pages 81-110.

Anderson **(1987)** P., Hankin C.L., Kelly P.H.J., Osmon P.E., Shute M.J. 'COBWEB-2: Structured Specification of a Wafer Scale Supercomputer' In *Proceedings of Parallel Architectures and Languages Europe, (PARLE '87),* LNCS **258**, Springer-Verlag, 1987, pages 51-67.

Annot **(1987)** J.K., Twist R.A.H. van. 'A Novel Deadlock Free and Starvation Free Packet Switching Communication Processor'. Philips Research Laboratories. Eindhoven. In *Proceedings of Parallel Architectures and Languages Europe (PARLE '87)*. LNCS **258**, Vol. I, Springer-Verlag, 1987, pages 68-85.

Appel **(1987)** A.W. 'Garbage Collection Can Be Faster Than Stack Allocation' In *Information Processing Letters*, **25**(4), 1987, pages 275-279.

Appel **(1994)** A.W. *An Empirical and Analytic Study of Stack vs. Heap Cost for Languages with Closures*, Technical Report CS-TR-450-94, Princeton University, 1994.

Augustsson **(1984)** L.A. 'A compiler for lazy ML' In *Proceedings of the ACM Symposium on LISP and Functional Programming*, Austin, Texas, ACM press, pages 218-227.

Augustsson **(1989)** L., Johnsson T. 'Parallel Graph Reduction with the $<v,G>$ - machine'. In *Proceedings of Functional Programming Languages and Computer Architecture, (FPCA '89)*, London, U.K. ACM press 1989, pages 202-213.

Backus **(1978)** J. 'Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs' In *Communications of the ACM*, **21**(8), 1978, pages 613-641.

Barendregt **(1987)** H.P., Eekelen M.C.J.D. van, Glauert J.R.W., Kennaway J.R., Plasmeijer M.J. and Sleep M.R. 'Term Graph Rewriting' In Bakker J.W. de, Nijman A.J., and Treleaven P.C., editors. *Proceedings of Parallel Architectures and Languages Europe (PARLE '87),* LNCS **259**, Vol. II, Springer-Verlag, 1987, pages 141-158.

Barendregt **(1992)** H.P., Beemster M., Hartel P.H., Hertzberger L.O., Hofman R.F.H., Langendoen K.G., Li L.L., Milikowski R., Mulder J.C., Vree W.G. *Programming Clustered Reduction Machines*. Technical report CS-92-05, Dept. of. Comp. Sys, University of Amsterdam, 1992.

Barendsen **(1992)** E. and Smetsers J.E.W. *Graph Rewriting and Copying.* Technical Report No. 92-20, Faculty of Mathematics and Computer Science, University of Nijmegen, the Netherlands, 1992.

Barendsen **(1993)** E. and Smetsers J.E.W. 'Conventional and Uniqueness Typing in Graph Rewrite Systems' (extended abstract). In Shyamasundar R.K. editor. *Proceedings of Foundations of Software Technology and Theoretical Computer Science,* LNCS **761**, Springer-Verlag, 1993, pages 41-51.

Barendsen **(1995-a)** E. and Smetsers J.E.W. 'Uniqueness Type Inference' In Hermenegildo M. and Swierstra S.D. editors. *Proceedings of the Seventh International Symposium on Programming Languages: Implementations, Logics and Programs,* LNCS **982**, Springer-Verlag 1995, pages 189-206.

Barendsen **(1995-b)** E. and Smetsers J.E.W. 'A Derivation System for Uniqueness Typing' In *Proceedings of joint COMPUGRAPH-SEMAGRAPH workshop on Graph Rewriting and Computation (SEGRAGRA '95),* Volterra, Pisa, Italy, Electronic Notes in Theoretical Computer Science, **1**, Elsevier, 1995.

Bertsekas **(1987)** D., Gallager R. *Data Networks*. Prentice-Hall International 1987.

Bevan **(1987)** D.I. 'Distributed garbage collection using reference counting', In *Proceedings of Parallel Architectures and Languages Europe (PARLE '87)*. LNCS **259**, Vol II, Springer-Verlag, 1987, pages 176-187.

Blelloch **(1993)** G.E., Chatterjee S., Hardwick J.C. , Sipelstein J., and Zagha M. 'Implementation of a Portable Nested Data-Parallel Language' In *Principles and Practice of Parallel Programming (PPoPP),* ACM press, 1993, pages 102-111.

Böhm **(1989)** A.P.W., Sargeant J. 'Code Optimisation for Tagged Token Dataflow Machines', In *IEEE Transactions on Computers*, **38**(1), January 1989.

Bratvold **(1993)** T.A. 'A Skeleton-Based Parallelising Compiler for ML'. In *Proceedings of the fifth International Workshop on the Implementation of Functional Languages*, Technical Report 93-21, Faculty of Mathematics and Computer Science, University of Nijmegen, the Netherlands, 1993, pages 23-33.

Bratvold **(1994)** T.A. 'Parallelising a Functional Program Using a List-Homomorphism Skeleton'. In *Proceedings of Parallel Symbolic Computation, (PASCO '94)*, Hagenberg/Linz, Austria, World Scientific, 1994, pages 44-53.

Bülk **(1993)** T., Held A., Kluge W., Pantke S., Rathsack C., Scholz S., Schröder R. 'Preliminary Experience with a π-RED+ Implementation on an nCUBE/2 System', In *Proceedings of the fifth International Workshop on the Implementation of Functional Languages,* Technical Report 93-21, Faculty of Mathematics and Computer Science, University of Nijmegen, the Netherlands, 1993, pages 101-113.

Burks **(1946)** A.W., Goldstine H.H. and Neuman J. von. 'Preliminary discussion of the logical design of an electronic computing instrument' In *John von Neumann, Collected Works*, **5**, pages 35-79.

Cann **(1992)** D. 'Retire Fortran? A debate rekindled' In *Communications of the ACM*, **35**(8), 1992, pages 81-89.

Church **(1936)** A. and Rosser J.B. 'Some properties of conversion' In *Trans American Mathematical Society*, **39**, pages 472-482.

Clarke **(1991)** L., Wilson G. 'Tiny: an efficient routing harness for the Inmos transputer'. In *Concurrency: Practice and Experience*, Vol. **3**(3), 1991, pages 221-245.

Cohen **(1981)** J. 'Garbage collection of linked data structures' In *Computing Surveys*, **13**(3), pages 341-367.

Cole **(1989)** M. 'Algorithmic Skeletons: Structured Management of Parallel Computation'. *Research Monographs in Parallel and Distributed Computing*. Pitman/MIT, 1989.

Curry **(1958)** H.B. and Feys R. *Combinatory Logic.* **1**. Amsterdam: North-Holland 1958.

Danelutto **(1993)** M. and Pelagatti S. 'Parallel Implementation of FP using a Template-based approach'. In *Proceedings of the fifth International Workshop on the Implementation of Functional Languages*, Technical Report 93-21, Faculty of Mathematics and Computer Science, University of Nijmegen, the Netherlands, 1993, pages 7-21.

Darlington **(1981)** J. and Reeve M.J. 'ALICE: A Multiple-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages' In *Functional Programming Languages and Computer Architecture, (FPCA '81)*, 1981, pages 65-76.

Darlington **(1993)** J., Field A.J., Harrison P.G., Kelly P.H.J., Sharp D.W.N., Wu Q., and While R.L. 'Parallel Programming Using Skeleton Functions' In *Parallel Architectures and Languages Europe, (PARLE '93)*, LNCS **694,** Springer-Verlag, 1993, pages 146-160.

Debbage **(1991)** M., Hill M., Nicole D. *Virtual Channel Router Version 2.0 User Guide*. Technical Report, University of Southampton, U.K. October 1991.

Eekelen **(1991)** M.C.J.D. van, Plasmeijer M.J., Smetsers J.E.W. 'Parallel Graph Rewriting on Loosely Coupled Machine Architectures' In Kaplan, Okada, editors. *Proceedings of the Workshop on Conditional and Typed Rewriting Systems, (CTRS '90).* LNCS **516**, Springer-Verlag, 1991, pages 354-369.

Eekelen **(1993)** M.C.J.D. van, Huitema H.S., Nöcker E.G.J.M.H., Plasmeijer M.J., and Smetsers J.E.W. *Concurrent Clean Language Manual - Version 0.8*. Technical Report No. 93-13, Faculty of Mathematics and Computer Science, University of Nijmegen, the Netherlands, 1993.

Gentleman **(1978)** W.M. 'Some Complexity Results for Matrix Computations on Parallel Processors', In *Journal of the ACM*, vol. **25**, 1978, pages 112-115.

George **(1989)** L. 'An Abstract Machine for Parallel Graph Reduction'. In *Proceedings of Functional Programming Languages and Computer Architecture, (FPCA '89)*, London, U.K., ACM press 1989, pages 214-229.

Girard **(1987)** J-Y. 'Linear Logic' In *Theoretical Computer Science*, **50**, pages 1-102.

Glaser **(1985)** H.W. and Thompson P. 'Lazy garbage collection' *Software Practice & Experience*, **17**(1), pages 1-4.

Goldsmith **(1993)** R., McBurney D.L. and Sleep M.R. 'Parallel Execution of Concurrent Clean on ZAPP' In Sleep M.R., Plasmeijer M.J., Eekelen M.C.J.D van, editors. *Term Graph Rewriting: Theory and Practice*, Wiley, 1993, chapter 21.

Groningen **(1991)** J.H.G van., Nöcker E.G.J.M.H. and Smetsers J.E.W.. 'Efficient Heap Management in the Concrete ABC Machine' In Glaser, Hartel editors. *Proceedings of the Third International Workshop on the Implementation of Functional Languages on Parallel Architectures*. Technical Report Series CSTR 91-07, University of Southampton, U.K., 1991.

Groningen **(1992)** J.H.G. van 'Some Implementation Aspects of Concurrent Clean on Distributed Memory Architectures', *Proceedings of the Fourth International Workshop on the Parallel Implementation of Functional Languages*, Aachener Informatik-Berichte Nr. 92-19, Fachgruppe Informatik, RWTH Aachen, Germany, 1992.

Groningen **(1993)** J.H.G. van 'Optimising Mark Scan Garbage Collection' In *Proceedings of the fifth International Workshop on the Implementation of Functional Languages*, Technical Report 93-21, Faculty of Mathematics and Computer Science, University of Nijmegen, the Netherlands, 1993, pages 185-192.

Hammond **(1995)** K., Mattson J.S. Jr., Partridge A.S., Peyton Jones S.L. and Trinder P.W. 'GUM: a portable parallel implementation of Haskell' In *Proceedings of the Workshop on the Implementation of Functional Languages '95*, Båstad, Sweden, September 13-15 1995, pages 259-280.

Hankin **(1985)** C.L., Osmon P.E., Shute M.J. 'COBWEB - a combinator reduction architecture' *In Proceedings of Functional Programming Languages and Computer Architecture, (FPCA '85)*, LNCS **201**, Springer-Verlag 1985, pages 99-112.

Harper **(1986)** R., MacQueen D. and Milner R. *Standard ML* Internal Report ECS-LFCS-86-2, Edinburgh University 1986.

Harrison **(1986)** P.G. and Reeve M.J. 'The Parallel Reduction Machine, Alice'. In *Graph Reduction*, LNCS **279**, Springer-Verlag 1986, pages 181-202.

Haynes **(1984)**, C.T. and Friedman D.P. 'Engines Build Process Abstractions' In *the ACM Conference on Lisp and Functional Programming*, 1984.

Hill **(1993)** J.M.D. 'The AIM is laziness in a data-parallel language' In K. Hammond, and J. T. O'Donnell, editors. *Glasgow Functional Programming workshop*, Springer-Verlag WICS, 1993.

Hoare **(1978)** C.A.R. 'Communicating Sequential Processes' *Communications of the ACM*, **21**(8), pages 666-677.

Hudak **(1986)** P. and Smith L. 'Para-functional Programming: A Paradigm for Programming Multiprocessor Systems' In *ACM POPL*, January 1986, pages 243-254.

Hudak **(1992)** P., Peyton Jones S., Wadler Ph., Boutel B., Fairbairn J., Fasel J., Hammond K., Hughes J., Johnsson Th., Kieburtz D., Nikhil R., Partain W. and Peterson J. 'Report on the programming language Haskell' *ACM SigPlan Notices*, **27**(5), pages 1-164.

INMOS **(1988)**. *Transputer instruction set*. Prentice-Hall 1988.

INMOS **(1988)**. *Transputer reference manual*. Prentice-Hall, 1988.

Keller **(1984)** R.M., Lin F.C.H. and Tanaka J. 'Rediflow Multiprocessing' In *IEEE Compcon*, February 1984, pages 410-417.

Kernighan **(1978)** B, Ritchie W. and Dennis M. *The C Programming Language*, Englewood Cliff NY: Prentice-Hall 1978.

Kesseler **(1990)** M.H.G. 'Concurrent Clean on Transputers', In *Proceedings of the Second Workshop of ESPRIT Parallel Computing Action (PCA)*, ISPRA, Italy, 1990.

Kesseler **(1991)** M.H.G. 'Implementing the ABC machine on transputers', In H. Glaser, P. Hartel, editors. *Proceedings of the third International Workshop on Implementation of Functional Languages on Parallel Architectures*, Technical Report 91-07, University of Southampton, U.K. 1991, pages 147-192.

Kesseler **(1992)** M.H.G. 'Communication issues regarding parallel functional graph rewriting', In H. Kuchen, R. Loogen, editors. *Proceedings of the fourth International Workshop on the Implementation of Functional Languages on Parallel Architectures*, Aachener Informatik-Berichte Nr. 92-19, Fachgruppe Informatik, RWTH Aachen, Germany, 1992.

Kesseler **(1993-a)** M.H.G. 'The Class Transputer Router', In V. Malyshkin, editors. *Parallel Computing Technologies, (PaCT'93)*, vol. I, NT-Centre, Obninsk, 1993, pages 235-250.

Kesseler **(1993-b)** M.H.G. 'Efficient routing using Class Climbing', In R. Grebe, J. Hektor, S. C. Hilton, M. R. Jane, P. H. Welch, editors. *Transputer Applications and Systems, World Transputer Congress '93*, Aachen, vol. 2, IOS Press, 1993, pages 830-846.

Kesseler **(1994-a)** M.H.G. 'Uniqueness and Lazy Graph Copying - Copyright for the Unique', In *Proceedings of the 6th International Workshop on the Implementation of Functional Languages*, University of East Anglia, Norwich, UK, 1994.

Kesseler **(1994-b)** M.H.G. 'Reducing Graph Copying Costs - Time to Wrap it Up'. In *Proceedings of Parallel Symbolic Computation, (PASCO '94)*, Hagenberg/Linz, Austria, World Scientific, 1994, pages 244-253.

Kesseler **(1995)** M.H.G. 'Constructing Skeletons in Clean - The Bare Bones', In *Proceedings of High Performance Functional Computing (HPFC '95)*, Denver, Colorado, CONF-9504126, Lawrence Livermore National Laboratory, 1995, pages 182-192.

Kingdon **(1991)** H., Lester D. R., Burn G.L. 'The HDG-machine: a Highly Distributed Graph-Reducer for a Transputer Network'. *The Computer Journal*, **34**(4), 1991, pages 290-301.

Koopman **(1990)** P.W.M., Eekelen M.C.J.D. van, Nöcker E.G.J.M.H., Smetsers S., Plasmeijer M.J. *The ABC-machine: A Sequential Stack-based Abstract Machine For Graph Rewriting*. Technical Report No. 90-22, Faculty of Mathematics and Computer Science, University of Nijmegen, the Netherlands, 1990.

Kuchen **(1994)** H., Plasmeijer R., and Stoltze H. 'Distributed Implementation of a Data Parallel Functional Language' In *Parallel Architectures & Languages Europe, (PARLE '94)* , pages 464-477, LNCS **817**, Springer Verlag, 1994.

Langendoen **(1993)** K. *Graph Reduction on Shared Memory Multiprocessors* PhD. Thesis, University of Amsterdam, the Netherlands, 1988.

Lester **(1993)** D.R. 'Distributed Garbage Collection of Cyclic Structures' In Hammond K., and O'Donnell J.T., editors. *Glasgow Workshop on Functional Programming*, Springer-Verlag WICS, 1993, pages 156-169.

Loogen **(1989)** R., Kuchen H., Indermark K., Damm W. 'Distributed Implementation of Programmed Graph Reduction'. In *Proceedings Parallel Architectures and Languages Europe (PARLE '89)*, LNCS **365/366**, Springer-Verlag, 1989, pages 136-157.

Magó **(1989)** G.A. and Stanat D.F. 'The FFP Machine' In *High-Level Language Computer Architectures*, 1989, pages 430-468.

Maranget **(1991)** L. 'GAML: a Parallel Implementation of Lazy ML'. In *Proceedings of Functional Programming Languages and Computer Architecture, (FPCA '91)*, LNCS **523**, Springer-Verlag, 1991, pages 102-123.

McBurney **(1987)** D.L. and Sleep M.R. 'Transputer-Based Experiments with the ZAPP Architecture' In *Parallel Architectures and Languages Europe, (PARLE '87)*, LNCS **258**, Springer-Verlag, 1987, pages 242-259.

McCarthy **(1960)** J. 'Recursive functions of symbolic expressions and their computation by machine' *Communications of the ACM*, **4**, pages 184-195.

McGraw **(1985)** J. SISAL: *Streams and Iterations in a Single-Assignment Language, Reference Manual version 1.2*. Manual M-146, Revision 1, Lawrence Livermore National Laboratory, 1985.

Minsky **(1963)** M.L. *A Lisp garbage collection algorithm using serial secondary storage*. Project MAC Memo 58 (rev.), Massachusetts Institute of Technology, 1963.

Nöcker **(1991-a)** E.G.J.M.H., Smetsers J.E.W., Eekelen M.C.J.D. van, Plasmeijer M.J. 'Concurrent Clean', In *Parallel Architectures and Languages Europe, (PARLE '91)*. LNCS **505**, Vol. II, pages 202-219.

Nöcker **(1991-b)** E.G.J.M.H., Plasmeijer M.J., Smetsers J.E.W. 'The PABC Machine' In *Proceedings of the Third International Workshop on Implementations of Functional Languages on Parallel Architectures*. Technical Report Series, CSTR 91 91-07, University of Southampton, U.K, 1991.

Nöcker **(1993-a)** E.G.J.M.H. 'Strictness Analysis using Abstract Reduction' In *Proceedings of Functional Programming Languages and Computer Architecture*. Copenhagen, Denmark, ACM Press, 1993, pages 255-265.

Nöcker **(1993-b)** E.G.J.M.H. and Smetsers J.E.W. 'Partially Strict Non-Recursive Data Types' In *Journal of Functional Programming*, **3**(2), 1993, pages 191-215.

Nöcker **(1993-c)** E.G.J.M.H. 'Efficient Parallel Functional Programming - Some Case Studies'. In *Proceedings of the fifth International Workshop on the Implementation of Functional Languages*, Technical Report 93-21, Faculty of Mathematics and Computer Science, University of Nijmegen, the Netherlands, 1993, pages 51-67.

Peyton Jones **(1987)** S.L. , Clack C., Salkild J., Hardie M. 'GRIP - a High Performance Architecture for Parallel Graph Reduction'. In *Proceedings of Functional Programming Languages and Computer Architecture (FPCA '87)*. LNCS **274**, Springer-Verlag 1987, pages 98-112.

Peyton Jones **(1989-a)** S.L. and Salkild, J. 'The spineless tagless G-machine' In *Proceedings of Functional Programming Languages and Computer Architectures, (FPCA '89)*, Reading MA, Addison-Wesley 1989, pages 184-201.

Peyton Jones **(1989-b)** S.L., Clack, C., Salkild, J. 'High Performance Parallel Graph Reduction'. In *Proceedings of Parallel Architecture and Languages Europe (PARLE '89)*, LNCS **365/366**, Springer-Verlag 1989, pages 193-206.

Peyton Jones **(1989-c)** S.L. 'Parallel Implementations of Functional Programming Languages', In *Computer Journal*, **32**(2), 1989, pages 175-186.

Plasmeijer **(1993)** M.J., Eekelen M.C.J.D. van *Functional Programming and Parallel Graph Rewriting*. Addison Wesley, 1993.

Plasmeijer **(1995)** M.J., Eekelen M.C.J.D. van *Clean 1.0 Language Report*, Technical Report, in preparation, University of Nijmegen, the Netherlands.

Richards **(1985)** H. *An Overview of Burroughs NORMA* Technical report, Austin Research Centre, Burroughs Corp., January 1985.

Sansom **(1991)** P.M.. 'Dual-mode garbage collection' In *Proceedings of the Third International Workshop on Implementations of Functional Languages on Parallel Architectures*. Technical Report Series, CSTR 91 91-07, University of Southampton, U.K., 1991.

Skillicorn **(1992)** D.B. 'The Bird-Meertens Formalism as a Parallel Model' In *NATO ARW "Software for Parallel Computation"*, 1992.

Smetsers **(1989)** J.E.W. *Compiling Clean to Abstract ABC-Machine Code*. Technical Report 89-20, Faculty of Mathematics and Computer Science, University of Nijmegen, the Netherlands, 1989.

Smetsers **(1991)** J.E.W., Nöcker E.G.J.M.H., Groningen J.H.G. van, Plasmeijer M.J. 'Generating Efficient Code for Lazy Functional Languages'. In *Proceedings of Functional Programming Languages and Computer Architecture, (FPCA '91)*, LNCS **523**, Springer-Verlag, 1991, pages 592-617.

Smetsers **(1993)** J., Barendsen E., Eekelen M.C.J.D. van, Plasmeijer M.J. 'Guaranteeing Safe Destructive Updates through a Type System with Uniqueness Information for Graphs'. In Schneider H.J., Ehrig H. editors *Proceedings of the Workshop on Graph Transformations in Computer Science*. LNCS **776**, Springer-Verlag, 1993, pages 358-379.

Son **(1991)** N. T., Paker Y. 'Adaptive Deadlock-free Packet Routeing in Transputer-based Multiprocessor Interconnection Networks'. In *The Computer Journal*, **34**(6), 1991, pages 493-502.

Stallings **(1988)** W. *Data and Computer Communications*. Macmillan Publishing Company, 1988.

Stoye **(1985)** W.R. *The Implementation of Functional Languages using Custom Hardware*, Ph.D. thesis, University of Cambridge, 1985.

Turner **(1985)** D.A. 'Miranda: a non-strict functional language with polymorphic types' In Jouannaud, J.P. editor. *Proceedings of Functional Programming Languages and Computer Architecture (FPCA '85)*, LNCS **201**, Springer-Verlag 1985, pages 1-16.

Wadler **(1990)** Ph. 'Linear types can change the world!' In Broy, M., Jones, C.B., editors. *Programming Concepts and Methods*, Amsterdam, North-Holland 1990.

Watson **(1986)** P. and Watson I. 'Graph Reduction in a Parallel Virtual Memory Environment'. In *Graph Reduction*, LNCS **279**, Springer-Verlag, 1986, pages 265-214.

Watson **(1987)** P. and Watson I. 'Evaluation of Functional Programs on the Flagship Machine'. In *Proceedings of Functional Programming Languages and Computer Architecture (FPCA '87)*. LNCS **274**, Springer-Verlag, 1987, pages 80-97.

Watson **(1989)** P., Watson I. 'An efficient garbage collection scheme for parallel computer architectures', In *Proceedings of Parallel Architectures and Languages Europe (PARLE '87)*. LNCS **259**, Vol II, Springer-Verlag, 1989, pages 432-443.

Watson **(1988)** I., Woods V., Watson P., Banach R., Greenberg M., Sargeant J. 'Flagship: A Parallel Architecture for Declarative Programming', In *15th IEEE/ACM symp. Computer Architecture*. Honolulu, Hawaii. SIGARCH newsletter, **16**(2), 1988, pages 124-130.

Wong **(1992)** W.F. and Yuen C.K. 'A Model of Speculative Parallelism' In *Parallel Processing Letters*, **2**(3), 1992, pages 265-272.

Index

Samenvatting

Dit proefschrift gaat over gereedschap. En zoals velen zullen beamen, valt zonder goed gereedschap geen degelijk huis te bouwen. Zo is het ook binnen de informatica: men heeft goede gereedschappen nodig om degelijke computerprogramma's te maken.

De beperkingen van de mens en de machine

Eenieder die wel eens met computers te maken heeft gehad zal opmerken dat de computerapparatuur (de hardware) wel steeds sneller wordt, maar dat het nog steeds slecht gesteld is met de kwaliteit van computerprogrammatuur (de software). Programma's geven soms foute antwoorden, zijn te traag en lopen af en toe gewoon 'vast'. Dit wordt ook wel de *software-crisis* genoemd.

Nu kan men beweren dat programmeurs hun werk niet goed doen, maar dit lost de software-crisis niet op. Beter is het te onderkennen dat zowel de computers zelf als de computerprogramma's steeds complexer worden en dat programmeurs klaarblijkelijk niet de juiste gereedschappen hebben om die complexiteit de baas te worden. In essentie is deze gedachte de basis van al het informatica-onderzoek: het hanteerbaar maken van complexe systemen.

Een duidelijk voorbeeld van een complex probleem is het programmeren van zogenaamde *parallelle computers*. Dit zijn machines die bestaan uit een groot aantal afzonderlijke computers die samenwerken om zo het werk sneller te doen. Het idee is simpel, maar het programmeren van dergelijke systemen is zo lastig dat het vaak voordeliger is om te wachten tot er snellere sequentiële computers zijn, dan om veel tijd te steken in de ontwikkeling van parallelle programma's. Dit is de voornaamste reden dat dergelijke parallelle machines nog niet wijd verspreid zijn.

Wat is er dan verkeerd aan het wachten op snellere sequentiële computers? Ten eerste zijn er problemen waarvoor *nu* een hogere berekeningssnelheid nodig is. Ten tweede kunnen sequentiële computers niet onbeperkt sneller worden. Op een gegeven moment loopt men tegen de snelheid van het licht aan. Nu al zijn er computers die meer dan honderd miljoen instructies per seconde kunnen uitvoeren. Bij zulke snelheden legt licht hooguit een paar meter af per instructie. Over enige tijd zal die afstand waarschijnlijk nog maar een paar centimeter zijn. Dan zal men informatie in een zeer klein computertje moeten stoppen, anders ligt het te ver weg om het nog op tijd bij een enkele instructie te krijgen. Snellere computers zullen daarom kleiner moeten zijn, maar oneindig klein is helaas onmogelijk.

Programmeertalen

Terug naar de gereedschappen voor het programmeren van computers. Een van de belangrijkste hulpmiddelen voor een programmeur is de *programmeertaal* waarin deze zijn programma schrijft. Met zo'n taal kan men heel precies beschrijven hoe een computer een bepaald probleem moet oplossen. Er zijn verschillende van die talen, maar ze zijn niet allemaal even begrijpelijk voor mensen. Gaandeweg worden er steeds begrijpelijker talen ontwikkeld. Echter, hoe begrijpelijker een taal voor de mens wordt, hoe onbegrijpelijker deze voor een computer wordt. Daarom moet zo'n taal vertaald (*gecompileerd*) worden naar een voor de computer duidelijke taal: de machinetaal (ofwel machinecode). Men spreekt ook wel over het *implementeren* van een taal op een computer. Voor talen van een hoog niveau is dit erg lastig, omdat die helemaal niet lijken op machinetaal. Dit proefschrift gaat over de implementatie van een bepaalde klasse programmeertalen van een zeer hoog niveau: de *functionele talen.* Daarbij beperken we ons tot één soort parallelle machine waarin elk computertje zijn eigen geheugen heeft: een parallelle machine met *gedistribueerd geheugen*.

Wanneer is een taal een functionele programmeertaal? Dit is het geval als een taal is opgebouwd uit *wiskundige functies.* De meest fundamentele eigenschap van zulke functies is dat hun betekenis volledig wordt bepaald door hun argumenten. Functies zijn *referentieel transparant*. In gewoon Nederlands betekent dit dat een functionele expressie altijd dezelfde betekenis heeft; het maakt niet uit waar en wanneer zo'n expressie gebruikt wordt. Traditionele *imperatieve* programmeertalen hebben die eigenschap niet. Zulke talen staan variabelen toe waaraan verschillende betekenissen kunnen worden toegekend. Zo'n variabele - de naam zegt het al - heeft dus niet altijd dezelfde betekenis.

Een functionele taal heeft belangrijke voordelen. Ten eerste is het mogelijk om eenvoudige wiskundige technieken te gebruiken om de correctheid van programma's te bewijzen. Verder kan men gemakkelijk redeneren over de betekenis van zo'n programma, omdat elke expressie dezelfde vaste betekenis heeft. En tenslotte is het mogelijk om expressies in een willekeurige volgorde uit te rekenen. Een andere volgorde verandert de betekenis van het programma namelijk niet. In het bijzonder wordt het zo mogelijk om verschillende expressies tegelijkertijd uit te rekenen. Dit houdt in dat het mogelijk wordt om programma's veel sneller uit te voeren op een parallelle machine, zonder het programma zelf wezenlijk te veranderen.

Het grootste nadeel van functionele talen is dat ze moeilijk zijn te vertalen naar machinetaal. In het bijzonder is dit lastig voor parallelle computers, en dan met name voor parallelle machines met gedistribueerd geheugen. Dit is vooral vervelend omdat het relatief gemakkelijk is om erg krachtige computers met gedistribueerd geheugen te maken, in tegenstelling tot parallelle computers met een gezamenlijk geheugen.

Implementatie

Dit proefschrift heeft een experimenteel karakter. Het beschrijft de implementatie van een concrete functionele taal - namelijk *Concurrent Clean* - op concrete parallelle *transputer* hardware. Deze implementatie is daadwerkelijk gebouwd en getest door metingen te verrichten. Deze werkwijze, alsmede de taal Concurrent Clean, de transputer hardware en

de problematiek van het implementeren van functionele talen worden beschreven in het eerste hoofdstuk.

De hoofdstukken 2 tot en met 5 beschrijven de technieken die nodig zijn om een functionele taal *volwaardig* te implementeren op transputer hardware. De onderwerpen van deze hoofdstukken zijn achtereenvolgens: de realisatie van efficiënte en betrouwbare communicatie over een transputer-netwerk, de logische structuur van de implementatie, het genereren van machinecode voor de transputer processor en tenslotte, het verdelen van willekeurige functionele expressies over het gedistribueerde geheugen (*het kopiëren van grafen)* en het automatisch verwijderen van expressies die niet meer gebruikt worden (*garbage collectie*). Een belangrijk verschil met andere onderzoeken op dit gebied is dat er geen enkele concessie is gedaan aan de algemeenheid van de functionele programmeertaal. Tegelijkertijd hebben we getracht om ook de implementatie zelf algemeen te houden door technieken te gebruiken die niet alleen toepasbaar zijn op transputer hardware, maar ook op andere computerarchitecturen. Dit heeft grote invloed gehad op het ontwerp - en de omvang - van de implementatie. Het blijkt echter dat zowel de algemeenheid van de taal, als die van de implementatie geen noemenswaardige nadelige invloed op de efficiëntie hebben indien men enige zorgvuldigheid in acht neemt bij het ontwerp.

Hoofdstuk 6 gaat dieper in op het verdelen van expressies over het gedistribueerde geheugen. Het blijkt dat sommige methoden niet te combineren zijn met belangrijke optimalisatietechnieken, zoals *uniciteits-typering.* Bovendien maken sommige verdeel-methoden het erg moeilijk voor programmeurs om te beredeneren hoe berekeningen verdeeld raken over de parallelle computer. Een nieuwe verdeelmethode genaamd *lazy normal form copying* biedt een oplossing voor beide problemen.

In hoofdstuk 7 zien we het belang van uniciteits-typering. Deze optimalisatietechniek werd in eerste instantie ontwikkeld om het mogelijk te maken grote compacte data-structuren te gebruiken die zeer gemakkelijk veranderd kunnen worden. Zulke structuren zijn essentieel voor het realiseren van efficiënte communicatie met de gebruiker (en de buitenwereld in het algemeen). We laten zien dat zulke datastructuren ook erg gemakkelijk verplaatst kunnen worden van de ene processor naar de andere. Ze zijn dan ook belangrijk voor het realiseren van efficiënte communicatie tussen computers onderling. Dit kan grote gevolgen hebben voor de snelheid waarmee sommige parallelle programma's worden uitgerekend.

Hoofdstuk 8 laat zien dat *hogere-orde functies* belangrijk zijn om functionele talen uit te breiden. Concurrent Clean heeft slechts een paar eenvoudige - maar fundamentele - *annotaties* die parallelle evaluatie van expressies bewerkstelligen. Voor het programmeren van ingewikkelde parallelle berekeningen heeft men veel van die simpele annotaties nodig. Dit maakt parallelle programma's soms onoverzichtelijk. Met hogere-orde functies kan men echter nieuwe constructies maken - *skeletten* - die een bepaalde soort ingewikkelde parallelle berekening beschrijven. Op die manier kunnen ingewikkelde - en zeer efficiënte - programma's op een eenvoudige manier geschreven worden.

De ontwikkeling van parallelle programma’s

Al met al is de implementatie van functionele programmeertalen op parallelle computers een lastig probleem, dat nog niet geheel is opgelost. Toch is al gebleken dat het programmeren van parallelle machines vele malen eenvoudiger is in een functionele taal, dan in een traditionele imperatieve taal. Tegelijkertijd moeten we opmerken dat het nog niet zo eenvoudig is als men wel zou willen. Functionele programma’s zijn dan wel zonder wezenlijke aanpassingen parallel uit te voeren, maar niet elk functioneel programma loopt zo snel als men zou verwachten. Dit komt omdat ook functionele programma’s vaak nog teveel sequentiële afhankelijkheden in zich bergen. Dit heeft veel te maken met de gebruikte datastructuren. Het lijkt dan ook noodzakelijk zich te richten op methoden die helpen bij het kiezen van de juiste datastructuren, of het mogelijk maken eenvoudig van structuur te veranderen. Misschien moeten concrete datastructuren pas geïntroduceerd worden in een laat stadium van de ontwikkeling van een programma.

Curriculum Vitae

Marcus Henricus Gerardus Kesseler

1968	Geboren te Heumen op 10 april
1980 - 1986	VWO, Elshofcollege te Nijmegen
1986 - 1990	Doctoraal Informatica, Katholieke Universiteit Nijmegen
1991 - 1995	Assistent in Opleiding (AIO), Computing Science Institute (CSI), Faculteit Wiskunde en Informatica, Katholieke Universiteit Nijmegen
1995	Dienstplichtig Reserve Officier Academisch Gevormd (ROAG), Frederikkazerne, Den Haag
1995 - heden	Wetenschappelijk medewerker bij Hollandse Signaalapparaten B.V. te Hengelo (Ov)