UNIVERSITY
*of*
GLASGOW

**Department of Computing Science**

# Granularity in Large-Scale Parallel Functional Programming

*Hans Wolfgang Loidl*

*A thesis submitted for a Doctor of Philosophy Degree in Computing Science at the University of Glasgow*

March 1998

# Abstract

This thesis demonstrates how to reduce the runtime of large non-strict functional programs using parallel evaluation. The parallelisation of several programs shows the importance of granularity, i.e. the computation costs of program expressions. The aspect of granularity is studied both on a practical level, by presenting and measuring runtime granularity improvement mechanisms, and at a more formal level, by devising a static granularity analysis.

By parallelising several large functional programs this thesis demonstrates for the first time the advantages of combining lazy and parallel evaluation on a large scale: laziness aids modularity, while parallelism reduces runtime. One of the parallel programs is the Lolita system which, with more than 47,000 lines of code, is the largest existing parallel non-strict functional program. A new mechanism for parallel programming, evaluation strategies, to which this thesis contributes, is shown to be useful in this parallelisation. Evaluation strategies simplify parallel programming by separating algorithmic code from code specifying dynamic behaviour. For large programs the abstraction provided by functions is maintained by using a data-oriented style of parallelism, which defines parallelism over intermediate data structures rather than inside the functions.

A highly parameterised simulator, GranSim, has been constructed collaboratively and is discussed in detail in this thesis. GranSim is a tool for architecture-independent parallelisation and a testbed for implementing runtime-system features of the parallel graph reduction model. By providing an idealised as well as an accurate model of the underlying parallel machine, GranSim has proven to be an essential part of an integrated parallel software engineering environment. Several parallel runtime-system features, such as granularity improvement mechanisms, have been tested via GranSim. It is publicly available and in active use at several universities worldwide.

In order to provide granularity information this thesis presents an inference-based static granularity analysis. This analysis combines two existing analyses, one for cost and one for size information. It determines an upper bound for the computation costs of evaluating an expression in a simple strict higher-order language. By exposing recurrences during cost reconstruction and using a library of recurrences and their closed forms, it is possible to infer the costs for some recursive functions. The possible performance improvements are assessed by measuring the parallel performance of a hand-analysed and annotated program.

# Contents

# List of tables

# List of figures

# Acknowledgements

First and foremost I would like to thank my supervisors, Kevin Hammond and Phil Trinder, for guiding my research over the course of my PhD and for helping me in avoiding the pitfalls on the way to obtaining a PhD degree. In particular, I am very grateful for initially getting the opportunity of doing research in Glasgow And I am indebted to both of my supervisors for the hand-holding done in the notoriously difficult writing-up stage.

I would like to thank the members of my viva panel, Greg Michaelson, John O'Donnell and David Watt, for the very constructive comments on how to improve my thesis.

I am indebted to members at the RISC-Linz institute, in particular to Bruno Buchberger, Hoon Hong and Wolfgang Schreiner, for giving me a great start into the academic world. And I am grateful to the Ministry of Science and Research, of the Republic of Austria for funding part of my PhD.

Finally my thanks to the whole functional programming group in Glasgow for providing such an active and stimulating environment.

# Declaration

I hereby declare that this thesis has been composed by myself, that the work herein is my own except where otherwise stated, and that the work presented has not been presented for any university degree before.

Sections 4.2 and 4.3 are revised versions of material published in (Trinder et al. 1998). Sections 4.4, 4.5 and 4.9.1 cover material to be published in (Loidl & Trinder 1997) and (Loidl et al. 1997), respectively. Section 4.6 is a revised version of material submitted for publication in (Loidl 1997). An earlier version of the material in Chapter 6 was published in (Loidl & Hammond 1996*a*).


- Trinder, P., Hammond, K., Loidl, H.-W. & Peyton Jones, S. (1998), Algorithm + Strategy = Parallelism, *Journal of Functional Programming* **8**(1).

- Loidl, H.-W. & Trinder, P. (1997), Engineering Large Parallel Functional Programs, *in* IFL'97 — International Workshop on the Implementation of Functional Languages, University of St. Andrews, Scotland, UK, September 10–12. To appear in LNCS.

- Loidl, H.-W., Morgan, R., Trinder, P., Poria, S., Cooper, C., Peyton Jones, S. & Garigliano, R. (1997), Parallelising a Large Functional Program; Or: Keeping LOLITA Busy, *in* IFL'97 — International Workshop on the Implementation of Functional Languages, University of St. Andrews, Scotland, UK, September 10–12. To appear in LNCS.

- Loidl, H.-W. (1997), LinSolv: A Case Study in Strategic Parallelism, *in* Glasgow Workshop on Functional Programming, Ullapool, Scotland, UK, September 15–17. Submitted for publication.

- Loidl, H.-W. & Hammond, K. (1996*a*), A Sized Time System for a Parallel Functional Language, *in* Glasgow Workshop on Functional Programming, Ullapool, Scotland, UK, July 8–10.


Hans Wolfgang Loidl

# Chapter 1

# Introduction

After decades of claiming that functional programming languages are well suited for implicitly-parallel execution, only a few systems have demonstrated this on a large scale. The research towards efficient implementations has revealed many problems in designing a parallel runtime-system that efficiently manages the generated parallelism without overwhelming the machine with bookkeeping overhead. The limited information provided by the programmer about the parallel execution of the program necessitates very sophisticated, and very general, runtime-system techniques.

One of the major strengths of functional languages is their clear and simple declarative semantics. From a compiler-design point of view this makes it possible to put theory to some practical use. For example static analyses are easily developed, which provide, at compile time, information about some runtime properties of the program. In the maturing sequential compiler technology for functional languages these analyses provide crucial information for program transformation steps, which represent the backbone of compiler optimisations. For the parallel execution of functional languages they can provide information to enable the runtime-system to manage the parallelism more efficiently.

This thesis investigates how to statically extract information about the granularity of potential parallel threads, i.e. the computation costs of these threads, and how to use this information in the runtime-system. In evaluating the importance of granularity for the efficiency of parallel program execution a set of large functional programs is studied. It transpires that a combinator-oriented approach towards exposing potential parallelism in the program leads to rather obfuscated code with intertwined behavioural and algorithmic code. To remedy this shortcoming this thesis contributes to a programming technique for separating these two kinds of code. This technique is used in the parallelisation of several programs, the largest of which consists of more than 47,000 lines of Haskell, making it the largest existing parallel non-strict functional program.

## 1.1 Parallel Lazy Functional Programming

Parallel computation offers an enticing picture of the performance that can be achieved by the next generation of computers: no longer is the program required to run on only one processor but it becomes possible to execute parts of the program on different processors. This enables the programmer to reduce the runtime of a program further by decomposing it into parallel components, either automatically or by hand. Potentially, it offers scalability in the performance of multiprocessors: in order to speed-up a machine it is only necessary to add new processors.

However, with most existing parallel programming models it is necessary to specify explicitly the decomposition of the program into parallel threads, the order of thread creation, the synchronisation, the communication between threads etc. In practice this often requires significant restructuring or even recoding of a sequential program. The root of this complication is the specification of an algorithm as a sequence of operations performed on a global store in an imperative programming model. In contrast, a declarative program does not specify such a sequence of operations. The compiler and the runtime-system are free to choose different orders of operations, or evaluation order, provided the semantics of the language is preserved. This opens up the possibility for an implicitly parallel execution of a declarative program where the programmer does not have to specify anything more than is needed for the sequential execution anyway.

Our programming model is therefore a combination of three models:

- *parallel programming* to reduce runtime by executing a program on several processors,
- *functional programming* to achieve a higher level of programming by abstracting over operational aspects,

- *non-strict programming* to increase modularity by decoupling control and definition.

The implementation model used in this thesis is parallel graph reduction. Section 2.3.1 discusses this model in more detail.

### 1.1.1 Parallel Programming

A parallel program reduces runtime by sharing the work to be done amongst many processors. To achieve such a reduction in runtime several threads, independent units of computation, are executed on different processors[1]. Introducing the concept of threads means that mechanisms for generating threads, synchronising threads, communicating data between threads, and terminating threads have to be established. We term these aspects of the program execution the dynamic behaviour of a parallel program. Clearly, the dynamic behaviour of a parallel program is significantly more complex than that of a sequential program.

Many existing parallel programming languages require the programmer to explicitly specify these aspects of parallel program execution. Objects specific to parallel execution, like semaphores and monitors, are used to describe synchronisation between threads. Managing these new objects, however, adds a new dimension of complexity to program development, for example the results of the parallel program might become non-deterministic, and especially the design of robust large-scale parallel systems becomes a daunting challenge.

The approach towards parallel computation advocated in this thesis is to let most of these resources be managed by the runtime-system in order to avoid the additional complexity for the programmer to handle these resources explicitly. All the programmer has to do is to *expose parallelism*, i.e. to identify parts of the program that may be usefully evaluated in parallel. This model is therefore termed one of semi-explicit parallelism. Ideally a compiler should automatically partition the program into parallel threads. If accurate strictness information is present this could be done by generating a parallel thread for every strict argument of an expression. However, the effects of different decompositions, or partitions, of the program into sequential

[1] We do not distinguish between complete heavy-weight threads, sometimes called tasks, and light-weight threads that can only exist within a task.

components are of special importance for the work presented in this thesis. Therefore the programmer is required to expose the potential parallelism in the program. In summary, our model offers the possibility of reducing the runtime by only exposing potential parallelism and without explicitly managing the parallel threads.

### 1.1.2 Functional Programming

Functional languages, as well as other declarative languages, describe what to compute without specifying the order in which to compute it. The exact evaluation order is only loosely defined by the data dependencies between expressions in the program. The compiler can choose any evaluation order of independent expressions. In particular, they can be evaluated in parallel. The semantic property that allows such a flexibility in the evaluation order is referential transparency, stating that the result of an expression does not change if a subexpression is replaced by another expression with the same result. For formal reasoning this allows to use the technique of replacing equals for equals. In the context of parallel computation this allows the compiler, or the runtime-system, to choose various orders of evaluation and to change them dynamically.

Based on this property of functional languages it is easy to implement a naive automatically parallelising compiler. For example, all strict arguments of a function call as well as the function body itself can be evaluated in parallel. However, the problem with this approach is the management overhead related to the vast amount of parallelism generated. Often the generated threads are too short to warrant an execution by a parallel thread altogether. Therefore, much effort has been put into increasing the length of these threads, which increases their granularity because each thread performs more computation.

This thesis studies how to increase the granularity of the generated threads and thereby improve the performance of the parallel program. A compile-time approach is taken, in which information about the granularity of potential parallel tasks is inferred at compile-time and forwarded, via automatically inserted annotations, to the runtime-system, which then uses this information in order to decide whether a parallel thread should be generated. This design naturally splits into one static component for inferring computation costs, a granularity analysis, and one dynamic component for using this information, granularity improvement mechanisms. It should be noted that the use of compile-time information from a static analysis does not amount to a static partitioning of the program. In our model the runtime-system is free to ignore parallelism. Thus, it is possible that different pieces of code that have been marked for parallel execution are actually merged into one thread by the runtime-system. In summary, we focus on functional languages because the lack of an explicit evaluation order specified in a program gives the compiler and the runtime-system a high degree of freedom in choosing a specific evaluation order. Although the use of implicit parallelism is not the immediate goal, this work makes some progress towards this long term goal.

### 1.1.3 Lazy Programming

An algorithm in a declarative language describes a property rather than a procedure. Executing the algorithm amounts to finding a solution for the property specified. This approach can be taken further to the level where values are bound to variables. The operational meaning of such a binding is to evaluate the expression. The declarative meaning, however, only identifies a variable with a value.

The idea of lazy evaluation, or more precisely of non-strict languages, is to decouple denotational definition from operational control. Defining the value of a variable does not mean that the definition has to be evaluated immediately. The definition only describes a property between a variable and a value in the program. The evaluation degree and the evaluation order are defined by the data dependencies in the program. This enables the reuse of the same variable in many different contexts, which examine different parts of the value. Thus, abstracting this control aspect out of the algorithm increases the modularity of programs.

There is an obvious tension between the goal of lazy evaluation, to abstract over control aspects of the code, and parallel computation, to enforce a parallel control structure of the code. Lazy evaluation tries to evaluate as small a portion of the result as possible, whereas parallel computation aims at generating independent threads of some minimal size. In order to achieve good parallel performance this means that at some places it may be necessary to specify how far a data structure should be evaluated, i.e. to specify its evaluation degree. Still, lazy evaluation is valuable for modular program design because this evaluation degree can be specified separately from the definition of the data structure itself. This encourages a data-oriented style of parallel programming, i.e. a style where the parallelism is specified over intermediate data structures rather than in the modules that generate these data structures. In the programming technique the parallel programming group at Glasgow has developed, evaluation strategies, this style of programming has proven to be extremely useful for large parallel programs.

The high degree of modularity provided by lazy languages is particularly important for the design of large programs. Furthermore, extremely time consuming programs, which would profit most from a reduction in runtime provided by parallel computation, are typically very large. Therefore, it is important that the language for parallelising the program supports modularity. Otherwise the gain in performance would have been bought with a loss in maintainability. In summary, the use of lazy evaluation decouples definition from control. This aides modularity and code re-use in a sequential model of computation. In a parallel model it also aides top down parallelisation of big programs by using data-oriented parallelism over intermediate data structures.

### 1.1.4 Relationship to Other Approaches for Parallel Programming

The approach towards parallelism taken by functional languages is in stark contrast to that taken by High Performance Fortran (HPF) (Rice 1993) and other parallel extensions of imperative languages. In parallel functional programming the programming language itself is unchanged. However, at certain points additional information is added to the program and used by the parallel runtime-system. This additional information only represents hints to the runtime-system that may be ignored rather than directives that must be obeyed. Therefore, the annotations do not change the semantics of the program. These annotations are in some sense analogous to register declarations in imperative languages that allow the programmer to add valuable operational information to the program but can be ignored by the compiler. It is interesting to note that many of these annotations, like register declarations, are nowadays rarely used and that most of the time automatic register allocation performed by the compiler is perfectly satisfactory for the programmer. Clearly, this state has not yet been achieved with parallelism annotations for functional languages. But the distinction between functional language features and operational annotations for parallelism enables a similar approach.

In contrast, parallel programs written in HPF-like languages aim at a near optimal usage of parallel machine resources. In doing so, they reveal low-level machine details and allow the program to specify details of the program execution leading to highly machine specific programs. As a result abstractions over primitive low-level constructs are evolving in the same way as high-level programming language constructs evolved out of common patterns of low-level instructions.

Based on these differences in the language design we consider parallel functional languages to be most useful for achieving moderate speed-up with only minimal changes in the code. Hopefully the necessary changes in the code that are still needed today can be reduced to zero with further progress towards implicit parallelism. HPF-like languages are more appropriate for applications in the supercomputing area where it is feasible to spend large programmer effort in restructuring code in order to get near optimal performance. However, we believe that the programming techniques used in our model, like data-oriented parallelism via non-strict data structures, can also be applied for this kind of languages in order to build high-level abstractions for certain kinds of parallelism.

## 1.2 The Dynamic Behaviour of Parallel Programs

The main reason for the complexity of writing parallel programs is the complex dynamic behaviour generated by a set of cooperating threads. In addition to the correctness of the sequential pieces of computation the timing of communication has to be considered in order to avoid deadlock situations and to guarantee both correctness and termination of the parallel program. Furthermore, the performance tuning of a parallel program requires a fine balance between several competing goals like creating many threads to use idle time of processors during the computation and limiting the number of generated threads to limit the bookkeeping overhead for the runtime-system.

Many parallel languages allow the programmer to control all these aspects of the dynamic behaviour. In our model, however, almost all of these details are hidden by the runtime-system. This design decision is based on the observation that the programmer is often overwhelmed with the complexity of writing a parallel program and explicitly managing the dynamic behaviour. In order to make such an semi-explicit approach feasible, the runtime-system has to make sophisticated decisions on how to manage the parallelism. For example, in our model the creation of parallel threads is based, to some extent, on the current load of a processor. The communication between threads is implicitly performed via reading and writing shared structures. The only extension necessary for specifying the parallelism in the program is a combinator that *exposes parallelism* called `par`. However, in order to get a more detailed control over the partitioning of the program into parallel threads it is often necessary to specify the *evaluation order* in an expression. This is done via adding `seq` combinators. Ideally, both kinds of combinators could be inserted into the program by an automatically parallelising compiler. However, first efficient runtime-system techniques to manage the parallelism have to be devised. The long term goal of this work is to automate this process of adding annotations describing the parallelism in the program.

One of the aspects of the dynamic behaviour is the granularity of a computation. By the granularity of a program expression we mean the computation costs of evaluating this expression. The inefficiency of fine-grained threads lies in the fact that they spend most of their computation on parallelism overhead like generating the thread or communicating with other threads. Historically, this has proven to be a severe problem for machines like ALICE (Darlington & Reeve 1981) and runtime-systems based on both graph-reduction (Hammond & Peyton Jones 1992, Hammond et al. 1994) and dataflow (Arvind & Nikhil 1990, Shaw et al. 1996). In order to mitigate this problem the programmer often tries to increase the granularity of the generated threads in the performance tuning stage of parallel program development. One goal of this thesis is to investigate how this process can be automated using statically-extracted information about the granularity of the generated threads. This information is used in the runtime-system to improve the performance of the parallel program without further information provided by the user.

This thesis studies granularity as one of the most important aspects of the dynamic behaviour of parallel program execution. However, it is, of course, not the sole important aspect of the dynamic behaviour. For example, the communication behaviour of the runtime-system determines the size of the graph structures that are sent within one unit of communication, determining the granularity of the communication. We have previously studied different fetching schemes in order to reduce the total communication overhead (Loidl & Hammond 1996*b*). Similarly, the scheduling mechanism is important to hide latency in a system involving a lot of communication. The data locality is an important property, which deserves further study, too.

## 1.3 Static Information about Dynamic Behaviour

One of the attractive features of functional languages for compiler optimisations is the fact that due to their clear semantic properties a lot of information about the program's dynamic behaviour can be inferred statically. The most important example of such a static analysis is strictness analysis, which detects expressions in a non-strict program that can be evaluated eagerly, and therefore more cheaply, without violating the semantics of the program. State-of-the-art compilers for non-strict functional languages like the Glasgow Haskell Compiler (GHC) (Peyton Jones et al. 1993, Peyton Jones 1996) heavily rely on the information provided by these analyses to perform a sequence of meaning preserving program transformations that improve the efficiency of the program.

Such statically-inferred information can also be exploited for parallel computation. However, because of the different dynamic behaviour of a parallel program additional information about the program execution is required. This thesis focuses on the aspect of granularity and presents a static granularity analysis, which is able to give an estimation of the computation costs of evaluating program expressions. Providing this additional information to the parallel runtime-system is an important step towards truly implicit parallelism for functional languages.

One important difference to classical analyses like strictness analysis, however, is the fact that granularity analysis has to infer information about an intensional property of the program execution. It can therefore be only correct with respect to an instrumented semantics, which itself models the property of interest. In this case computation costs are modelled as computation steps and inferred as an estimate for an upper bound. This indirect way of extracting information affects the quality of the result. However, in contrast to strictness analysis wrong granularity information will not affect the semantics of the generated program but only its performance. Therefore it is possible to design an analysis that sometimes makes guesses about computation costs.

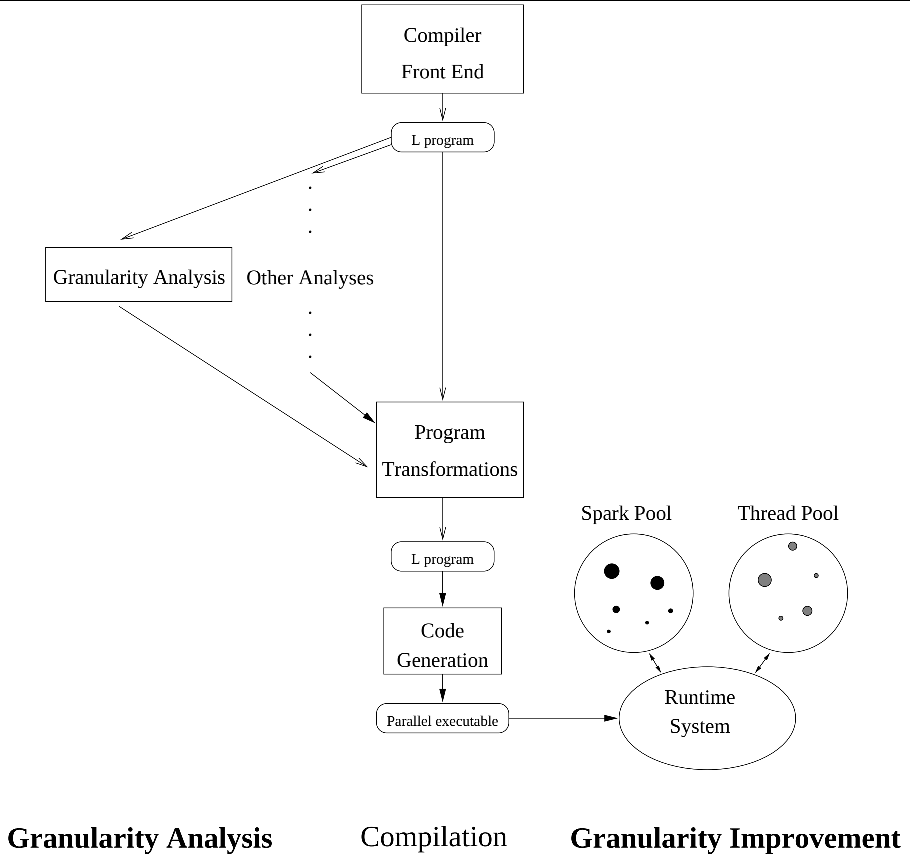


**Figure 1.1** Possible structure of a parallelising compiler

Figure 1.1 summarises a possible overall structure of a parallelising compiler. The front end of the compiler translates the input program into an intermediate language, called $\mathcal{L}$. This language is designed to be simple in order to ease later analysis and program transformation stages, operating on this language. The program transformation stages, which present the main part of the compiler, perform program optimisations and make use of the information provided by various static analyses such as granularity analysis to obtain information about the evaluation costs of program expressions. In the programming model used in this thesis parallelism annotations have to be present in the input program already. The program transformations can then add further information to the existing annotations. However, at this stage enough information is available to automatically insert parallelism annotations, if the goal is implicit parallelism. Finally, the code generation stage of the compiler produces a parallel executable. In the setup used in this thesis the parallel executable will be machine independent by using a runtime-system that hides details of the parallel architecture. As a further optimisation it would be possible to generate specialised code for particular parallel machines. The granularity improvement mechanisms that are developed in this thesis then make use of the additional granularity information attached to sparks and threads to make better scheduling decisions based on this additional information.

In summary, this thesis focuses on the parallel execution of non-strict functional programs that are annotated in order to expose potential parallelism. A parallel graph reduction model is used to implement the parallel execution of the program. In particular, this thesis tackles two parts in the structure shown in Figure 1.1: the granularity analysis and the granularity improvement mechanisms.

## 1.4 Contributions

This section gives a list of research contributions made in this thesis. A more detailed discussion of the contents of the contributions with a separation of the authorship of parts in the contributions is given at the end of the thesis in Section 7.2.

1. *Parallelisation of large lazy functional programs* (Loidl & Trinder 1997): This thesis demonstrates how to combine the advantages of lazy evaluation, in particular modularity, and of parallel evaluation, namely reduced runtime, on a large scale. In the parallelisation of a set of large algorithms the modularity provided by lazy evaluation helps to minimise the code changes required to improve the parallel performance of the program. The implementation includes both the design of parallel functional algorithms, such as LinSolv, as well as parallelising existing code, such as Lolita. With more than 47,000 lines of Haskell code Lolita is the largest existing parallel non-strict functional program. The programs demonstrate a crucially important aspect of strategic programming in the large, namely the separation of behavioural from algorithmic code.

2. *Highly parameterised, accurate simulator* (GranSim) (Hammond et al. 1995): The collaboratively developed GranSim simulator is of use for architecture-independent parallelisation as well as a testbed for the implementation of specific runtime-system features. Its robustness has been tested with large parallel applications. By being highly parameterised it is very flexible in the parallelisation

and tuning of functional programs. By being accurate and closely related to the parallel GUM runtime-system it encourages prototype implementations of specific runtime-system features. GranSim has been integrated into an engineering environment for parallel program development in order to facilitate the development and performance tuning of large programs. A set of visualisation tools has proven crucial for understanding the dynamic behaviour of GranSim and GUM programs. Primary contributions to GranSim made in this thesis include the design of the communication system, the implementation of an idealised simulation, and the integration of GranSim into GHC.

3. *Use and refinement of evaluation strategies* (Trinder et al. 1998): This thesis contributes to evaluation strategies by adding strategic function application and by providing some of the first uses of strategies. The latter in part drove the design of the current version of strategies. Strategic function application has proven very useful in large parallel applications such as Lolita. In particular, it supports data-oriented parallelisation, which achieves high modularity by decoupling the definition of a function from the specification of its parallelism.

4. *A static granularity analysis* (Loidl & Hammond 1996*a*): A granularity analysis for inferring upper bounds of computation costs in a simple strict higher-order language, based on existing analyses (Hughes et al. 1996, Reistad & Gifford 1994), is presented. The analysis is formulated as a subtype inference system. A detailed outline of an implementation is given and an extended cost reconstruction algorithm is developed. The analysis has not been implemented but measurements with a hand analysed program allow some assessment of the importance of the inferred information.

5. *Implementation and measurement of runtime-system features to improve parallel performance:* (Loidl & Hammond 1995): This thesis discusses several granularity improvement mechanisms the author has implemented in GranSim. Measurements studying their impact on the parallel performance of a set of test programs are provided. As a result moderate improvements in performance have been achieved for programs that are annotated with granularity information.

In addition to the major contributions above this thesis also makes less significant contributions towards a comparison of imperative and functional parallel programming by presenting results from parallel imperative implementations of three computer algebra algorithms in Section 4.7. Chapter 2 gives a detailed survey of several techniques for the parallel implementation of functional languages, going beyond the issues addressed in the main part of the thesis, and Sections 5.7 and 6.7 survey alternative approaches for improving granularity and for designing analyses extracting granularity information, respectively. In the examination of large programs other runtime-system aspects of the parallel execution of lazy functional programs have proven important. Different packing and rescheduling schemes have been implemented in GranSim, addressing the issue of efficient communication in a parallel graph reduction system (see Section 3.3.1). Details of the implementation and various measurements are presented elsewhere (Loidl & Hammond 1996*b*).

## 1.5 Thesis Structure

The structure of this thesis is as follows.

**Chapter 2** gives a survey of various approaches towards a parallel implementation of functional languages. In particular, this chapter describes details of the parallel graph reduction model that is used in this thesis and its relationship to other execution models. The discussion distinguishes key runtime-system issues for parallel program execution: the evaluation model, the storage management model, the communication model, and the load distribution mechanism.

**Chapter 3** gives a detailed description of the GranSim simulator that is developed in this thesis. GranSim is a flexible and accurate simulator for the parallel execution of Haskell programs. It supports both an idealised simulation and an accurate simulation modelling the characteristics of a particular architecture. In parallelising a set of large Haskell programs GranSim has been extensively used for developing and tuning the parallel code. In later chapters GranSim will be used as the platform for measurements on granularity.

**Chapter 4** discusses the parallelisation of several large lazy functional programs. This chapter first presents evaluation strategies, which have been developed in a group effort. Then three programs are discussed in detail: a parallel Alpha-Beta search algorithm, highlighting the interplay between lazy and parallel evaluation, LinSolv, a symbolic computation algorithm using infinite intermediate data structures, and

Lolita, a large natural language engineering system.

**Chapter 5** focuses on the aspect of granularity for the dynamics of parallel program execution. For a set of programs the granularity of the generated threads is measured. It is shown that by increasing the granularity the performance of the programs can be improved. Three different granularity improvement mechanisms are discussed and measured: explicit thresholding, priority sparking, and priority scheduling.

**Chapter 6** presents a static granularity analysis for a simple strict functional language. This analysis infers an upper bound for the number of computation steps needed to evaluate a program expression. The analysis is developed as an inference system together with an analysis for the size of program values. A detailed outline of a possible implementation is given, combining two existing analyses. Finally, a small test program is hand-analysed and the resulting annotated program is measured showing some performance improvements.

**Chapter 7** draws conclusions from the presented approach towards improving the performance of parallel lazy functional programs. It evaluates the importance of a structured approach towards program parallelisation, in particular for the performance tuning stage of parallel program development. And it identifies areas of future work, in particular for achieving the long term goal of truly implicitly parallel execution of functional programs.

# *Chapter 2*

# The Parallel Implementation of Functional Languages

**Capsule**

This chapter discusses several approaches towards a parallel implementation of functional languages. It starts with motivating the use of functional languages for parallel programming. Then it presents the basic ideas of popular models for the implementation of functional languages and evaluates how easily parallel evaluation can be expressed in these models. The main part of this chapter focuses on critical runtime-system issues and outlines several efficient implementation techniques. The following runtime-system issues are examined:

- the evaluation model,
- the storage management,
- the communication model, and
- load distribution.

In this thesis a *parallel graph reduction* model is used. The mechanisms for implementing the above runtime-system issues in this model are compared with possible alternatives. The overall discussion is based on an implementation on stock hardware rather than specialised hardware for functional programming.

## 2.1 Introduction

In assessing the quality of various kinds of programming languages the requirement of parallel execution usually complicates the language and therefore diminishes its value for large-scale program design. Not so with functional languages! The higher level of abstraction, compared to imperative languages, decouples the semantics of the language from operational considerations such as sequential or parallel evaluation. In particular the referentially transparent nature of functional languages allows various different ways of evaluating an expression. However, implementing an efficient system for parallel functional programming, consisting of an optimising compiler and a flexible runtime-system, has proven to be quite difficult.

Functional languages and their implementation have a rather long history. Whereas early models for implementing functional languages were defined on a rather low level, e.g. the SECD machine (Landin 1964), more recent models such as the graph reduction and the dataflow models present a far higher level of abstraction, allowing parallelism to be expressed naturally in this framework. However, when implementing such a model many runtime-system issues have to be tackled. The core of this chapter deals with the efficient implementation of these runtime-system issues on stock hardware. We do not consider special purpose hardware since the development on parallel hardware during the last years has shown a clear focus on general purpose machines.

The structure of this chapter is as follows. Section 2.2 discusses how functional languages can express parallelism in general, and which kind of model is used in this thesis. Section 2.3 outlines several models for implementing functional languages and evaluates how easily parallel evaluation can be expressed in these models. Section 2.4 focuses on key issues of the runtime-system for the efficient parallel implementation. Section 2.5 puts our model into the context developed thus far. Finally, Section 2.6 summarises aspects of our implementation model that have to be addressed in order to construct an efficient parallel evaluation of functional languages.

## 2.2 Principles of Parallel Functional Languages

This section discusses why functional languages are a good vehicle for writing parallel programs. It discusses some semantic issues that have an important impact on the parallel behaviour of the program, and connects them with runtime-system issues discussed in more detail in subsequent sections.

### 2.2.1 Why are Functional Languages Good for Parallelism?

With the advent of parallel machine architectures and their promise of far higher performance than it is possible for conventional architectures, the design of languages for parallel computation has become an important research topic. A key aspect in the design of parallel languages is the way that the parallel execution is described. *Imperative languages* traditionally extend the sequential model with explicitly handled threads to describe independent pieces of computation and messages to communicate data between these threads. If these notions remain visible to the programmer he has to cope with issues like possible deadlocks in the computation, the partitioning of the computation into components, and the placement of these computations onto the processors of the parallel machine. This adds a new dimension of complexity to the design of a parallel algorithm and distracts from the mathematical properties of the algorithm like its correctness.

Another approach, which restricts the generality of this message passing style of computation, has recently become extremely popular: synchronous parallel computing. The two best known models in this class are BSP (McColl 1996) and SPMD (Smirni et al. 1995). The idea in these models is to synchronise all communication in the system by either alternating between supersteps of computation and communication, or by using an implicit barrier for finishing all communication. This restriction enforces a certain structure of the parallel program. However, it also facilitates the performance evaluation of the program. Furthermore, the basic communication operation in these models, namely broadcast, can be implemented very efficiently on the latest parallel hardware. Here hardware realisation and programming model go hand in hand, similar to the success of RISC machines for sequential computation. However, usually the programmer still has to handle explicit threads and messages, which complicates the parallel program significantly compared to the sequential model. This thesis focuses on a higher-level approach of parallel programming, hiding most of these aspects in the runtime-system. It is, however, still possible to re-use existing lower-level code for specialised tasks.

In contrast, *functional languages* provide a higher level of abstraction by only specifying what to compute without specifying a sequence of instructions describing how to compute the result. As a result functional languages are referentially transparent, which implies that independent parts of the program can be evaluated in parallel. Thus, the language does not necessarily need to be extended to deal with parallel evaluation. In principle, the problem of parallelising an algorithm can be reduced to the problem of reducing data dependencies in the program — something that can be done via source-to-source program transformations in much the same way as program optimisations in sequential compilers. Reasoning about the correctness of such transformations is no more difficult than for standard transformations used in sequential optimising compilers. Furthermore, parallelism based on functional languages yields a *deterministic result*, and it is guaranteed to be the same result as in the sequential execution. There is no danger for deadlock in such a model, unless a program runs out of resources.

Of course, the higher level of abstraction also imposes some overhead on the execution. Therefore, an optimised parallel algorithm using lower level features like an imperative computation model and message passing for communication will usually result in a better performance of the algorithm. However, especially for large programs it is extremely difficult to work at such a low level of abstraction.

### 2.2.2 The Role of Strictness

This section discusses fundamental semantic properties of functional programming languages and their impact on the sequential and parallel evaluation of such languages. It focuses on strictness as the most important of these properties.

#### Definition of Strictness

One important semantic property of a programming language is the strictness of user defined functions. A function is strict if its result is undefined, whenever the its argument is undefined. A non-strict language is a language that permits the definition of non-strict functions. More formally, a function $f$ is strict if and only if

$$f\ \bot\ = \bot$$

where $\perp$ represents an undefined result (e.g. caused by a failing or non-terminating computation). A discussion of strictness is given for example in (Field & Harrison 1988)[Chapter 4].

One important advantage of non-strict over strict parallel languages is the ease of expressing producer/consumer parallelism in the former. In particular the coroutine nature of lazy evaluation avoids a barrier synchronisation between the producer and the consumer process. Following the terminology of Goldberg (1988*a*) this means, it is easy to express *vertical parallelism*, i.e. parallelism between a function and its argument, in a non-strict language. In contrast, strict languages tend to rely more on *horizontal parallelism*, parallelism between different arguments, which evaluates the arguments of a function in parallel. It should be noted that this form of parallelism can also be used in non-strict languages, namely for those argument positions in which the function is strict. A separate strictness analysis is needed to determine which arguments can be safely evaluated before the function itself is called.

In order to use a parallel function application, strictness information on user defined functions is needed, which ensures that creating parallel threads for each argument satisfies the non-strict semantics of the program. The resulting parallelism is called *conservative parallelism*, i.e. the values of all parallel threads are known to be needed in the computation. If non-strict arguments are evaluated in parallel, too, *speculative parallelism* is generated. Dealing with this kind of parallelism complicates the underlying evaluation model because it must be ensured that no process consumes all available resources and it should be possible to terminate processes. However, if this is guaranteed on runtime-system level then the parallel evaluation of all arguments in a function call satisfies the non-strict semantics, too. Although speculative parallelism is an important issue for parallel functional languages, it is not directly related to the main runtime-system aspect this thesis is investigating: granularity. Therefore, this thesis does not give an exhaustive survey of this particular branch of the field.

### Evaluation Mechanisms

This section briefly discusses possible evaluation mechanisms for functional languages. These definitions build on top of the notion of reduction in the lambda-calculus (Church 1941) and delta-reduction for built-in rules like basic arithmetic. The terminology of this chapter follows (Field & Harrison 1988)[Chapter 6].

**Definition 1 (redex)** *A* redex *(reducible expression) is an expression that can be reduced according to the rules of lambda-calculus or delta-reduction.*

Intuitively, a redex is an expression that can be immediately evaluated. To be more precise about the degree of evaluation several different normal forms can be distinguished.

**Definition 2 (weak head normal form)** *An expression is in* weak head normal form *(WHNF) if, and only if, it is a constant or if it is of the form*

$$f\ e_1\ \ldots\ e_n,\quad \textit{for some}\ 0 \leq n < \textit{arity of}\ f$$

*where $f$ is either a data constructor or function (primitive or user defined).*

Intuitively, evaluating an expression to weak head normal form means evaluating only the top level constructor. The expressions $e_1\ \ldots\ e_n$ may still contain redexes.

**Definition 3 (normal form)** *An expression is in* normal form *if it does not contain any redexes.*

An expression in normal form matches the intuitive notion of a value in the language. In an expression, which is not in normal form, the leftmost redex is the redex textually left to all other redexes and the outermost redex is the redex not contained in another redex. Based on these definitions and the two normal forms above it is possible to specify the reduction order yielding the two main evaluation mechanisms used in this thesis.

**Definition 4 (eager evaluation, call-by-value)** *An* eager evaluation *mechanism chooses in every reduction step the leftmost innermost redex and reduces it to weak head normal form.*

**Definition 5 (lazy evaluation, call-by-need)** *A* lazy evaluation *mechanism chooses in every reduction step the leftmost outermost redex and reduces it to weak head normal form. When substituting expressions for arguments no expression is duplicated, but they are shared in the reduced expression.*

The *evaluation transformer* (Burn 1987, Burn 1991*b*) approach for automatic parallelisation defines a whole set of such evaluation mechanisms, which are tuned to the strictness of the result that should be computed in the given context. It uses detailed strictness information obtained by a sophisticated strictness analysis to determine, given the demand on an expression, how far the components of the expression have to be evaluated. Thus, all components can be safely evaluated in parallel to the degree determined by the evaluation transformer. However, this requires the generation of several variants of the code for each function, specialised to the particular context in which it is used. This approach has been used by Burn (1991*a*), in the distributed-memory HDG machine (Kingdon et al. 1991), in the PAM machine (Loogen et al. 1989), in Rushall's parallel implementation of the Spineless G-machine on top of a virtual shared-memory KSR1 machine (Rushall 1995), and in the shared-memory EQUALS system (Kaser et al. 1997).

#### Beyond Strictness

In order to preserve the semantics of the program, strictness information is needed for implicit parallelisation in order to decide which arguments can be safely evaluated in parallel. However, more information about dynamic properties of the program is useful in order to extract efficient parallelism. In particular, *granularity information*, i.e. information about the size of a computation, is needed in order to decide whether it is worth paying thread creation and synchronisation overhead for computing an expression in parallel. This question is discussed in detail in later chapters. Chapter 5 shows that too fine granularity can deteriorate parallel performance and develops runtime-system mechanisms to increase granularity. Chapter 6 presents a granularity analysis for a simple strict, higher-order language to estimate the costs of an evaluation.

### 2.2.3 Language Support for Parallel Programming

The previous section has shown that it is possible to automatically parallelise a functional program by executing all strict arguments of a function call in parallel. Sharing and granularity information, if available, can be used to determine whether it is worthwhile creating a thread for a computation.

However, developing these analyses is a non-trivial problem. In fact, part of this thesis is devoted to the development of a simple granularity analysis for a small strict functional language. In the absence of a compiler that can automatically detect parallelism it is useful to make the information about potential parallelism and the size of the computation explicit in the language. In contrast to models of explicit parallelism, the sparking model used in this thesis only needs constructs for *exposing parallelism*. Creation of threads, synchronisation, and communication are all implicit in this model. Therefore, we call this a model of *semi-explicit parallelism*.

This section first discusses some features of lazy functional languages, which are of importance for the rest of this thesis. Then the basic constructs for parallelism in this language are described. Finally, a comparison with other approaches towards language support for parallel computation is given.

### Lazy Functional Programming

This section highlights the most important features of lazy functional languages that are of relevance for this thesis. An excellent general discussion of lazy functional programming is given in Bird & Wadler (1988).

A lazy evaluation mechanism, as defined in the previous section, will only evaluate an expression, if its value is required in the computation. This results in a *demand-driven* order of evaluation. An obvious advantage of this mechanism is that no unnecessary expressions will be evaluated. Another, even more important, aspect is the fact that the definition of a result is separated from its evaluation. Thus, it becomes possible to describe details of the evaluation, such as parallelism, without modifying the code that defines the result. This feature plays a crucial role in our technique for large-scale parallel programming and will be elaborated in detail in Section 4.3.

A very powerful feature provided by most functional languages is the availability of *higher-order functions*, i.e. functions that take other functions as arguments or that return a function as a result. Such higher-order functions can be used to express common patterns of computation. For example the Haskell prelude function `map` performs the same operation, given as a first argument, to every element of a list, given as the second argument. In the context of parallel computation, higher-order functions are a natural choice for expressing parallel behaviour. Indeed, our parallel programming technique makes heavy use of higher-order functions. However, in contrast to related approaches such as skeletons (Cole 1989), the parallelism is not restricted to a fixed set of higher-order functions.

Functional languages offer powerful constructs operating on *algebraic data types*. This encourages the construction of elaborate data structures such as lists or trees, which are best fit for expressing a certain algorithm. This facility is of particular importance in the area of symbolic computation where the data is typically non-numeric and highly-structured. With algebraic data types pattern matching is often used to simultaneously check the structure of a data item and to bind components to names.

For example the aforementioned `map` function is defined as follows in Haskell 1.2:

```
map           :: (a -> b) -> [a] -> [b]
map f []      = []
map f (x:xs) = f x : map f xs
```

The first line specifies the type of the function, which is useful for documentation of the code and as additional information for the compiler. In this case type variables `a` and `b` are used, to express that `map` is a polymorphic function, which can operate on any list provided the domain of the function `f` has the same type as the elements of the list provided as second argument. The result type will be a list with elements of the same type as the codomain of the function `f`. Note that all type variables are universally quantified to achieve this kind of polymorphism.

The next two lines perform pattern matching on the list argument. If this argument is non-empty it is constructed via the : operator with the arguments `x` and `xs`, which are used on the right hand side of the definition. Note that, because `f` is a function, map is a higher-order function. With this definition `map` can be used to translate all characters in a string into upper case characters via `map toUpper "hello"`. Being polymorphic it can be also used to, e.g. count the elements of all sub-lists in a given list of lists via `map length [[1],[1,2],[1,2,3]]`.

The above examples used Haskell prelude functions such as `toUpper c` for translating the character `c` into an upper case character and `length xs`, for computing the length of the list `xs`. Some other basic prelude functions that will be used in this thesis are `take n xs` for returning the first `n` elements of the list `xs`, `filter p xs` for returning a list of all elements of `xs` for which the predicate `p` evaluates to true, and `foldl f z` `xs` for combining, from left to right, all list elements of `xs` with the binary operator `f`, using `z` as the start value. The construct `xs!!n` extracts the `n`-th element from the list `xs`, and `f$x` applies the function `f` to the argument `x` (this construct is useful in a sequence of nested function applications in order to avoid nested parenthesis).

### GpH

This thesis uses GpH as a parallel functional programming language. GpH is an extension of the non-strict, purely functional programming language Haskell (Peterson et al. 1996). It is augmented with sequential and parallel combinators.

**Sequential Combinator:** The `seq` operator specifies the order of evaluating two expressions. The operational semantics of the expression ``e1 `seq` e2`` is as follows: first evaluate the expression `e1` then the expression `e2`. Both are evaluated to WHNF. It is an asymmetric combinator of type `seq :: a -> b -> b`, which returns the second argument as a result, i.e. the denotational semantics of `seq` is

$$\begin{aligned} \texttt{seq}\ \perp\ e_2 &= \perp \\ \texttt{seq}\ e_1\ e_2 &= e_2 \quad \textit{if}\ e_1 \neq \perp \end{aligned}$$

**Parallel Combinator:** The `par` operator introduces parallelism in the language. It also has the type `par :: a -> b -> b`. The operational semantics of the expression ``e1 `par` e2`` is as follows: first record that `e1` can be evaluated in parallel then evaluate `e2`. We christen the operation of recording the possibility of parallel evaluation to *spark* an expression. It is important to note that this is very different from creating a thread for evaluating the expression. Sparking an expression can be done very cheaply. In our model a pointer to an unevaluated expression is put into a spark pool, a special data structure maintained by the runtime-system. Furthermore, the sparking model defers the decision whether to create a thread or not to a later time. Details of these runtime-system issues are discussed in detail in Section 2.4. The denotational semantics of `par` is

$$\texttt{par}\ e_1\ e_2 = e_2$$

Note that `seq` is strict in its first argument, whereas `par` is non-strict in both arguments.

### Extensions of GpH

One important aspect of the work in this thesis is to propagate information about the program's behaviour to the runtime-system. The `par` construct can be seen as a way to propagate information about potential parallelism to the runtime-system. To give the programmer the possibility of specifying additional information about the parallel processes, several lower level constructs are provided. They take additional arguments and propagate this information to the runtime-system. The denotational semantics of these constructs is the same as for `par`.

**Global Parallelism:** The additional arguments in a `parGlobal n g s p x y` expression have the following meaning: `n` is the name of the spark, `g` represents the granularity of the computation, `s` represents the size of the result and `p` represents the degree of parallelism created during the evaluation of the expression. The latter is an estimate on the number of sparks generated in the expression `x`. All of these arguments are integers.

The GranSim simulator discussed in Chapter 3 currently only uses the information in the `n` and `g` fields. The former helps to distinguish sparks from different static spark sites. The latter is the main piece of information that is exploited via the granularity control mechanisms described in Section 5.5.

**Local Parallelism:** The `parLocal` construct, which takes the same arguments as `parGlobal`, enforces that the thread for the sparked expression, if it is created, will be started on the same processor where it was created. However, since the runtime-system may use thread migration, this does not mean that the thread has to remain on that processor throughout its computation. The main purpose of this construct is to improve data-locality between sparks that operate on the same data.

**Thread Placement:** The `parAt` construct is a generalisation of `parLocal`. It requires the thread to be generated on a specific processor, specified by an integer value. This assumes that the names of all processors form a sequence from 0 to some integer value $n$. This is an experimental feature that has been used in one parallel algorithm so far.

### Other Approaches

The semi-explicit approach for describing parallelism, which is used in this thesis, defers most of the control of the parallelism to the runtime-system. On the language level it is only necessary to provide constructs that expose parallelism. In the range from implicit to explicit models of parallel computation our model is therefore close to the implicit end. The following discussion locates the models that are discussed in more detail in this chapter on this range from implicit to explicit parallelism.

Some examples of fully implicit models are dataflow languages such as Id (Nikhil 1989), pH (Aditya et al. 1995) and SISAL (Böhm et al. 1991), evaluation transformers (Burn 1987), and data parallel languages such as NESL (Blelloch 1996). Algorithmic skeletons (Cole 1989) provide a set of higher-order functions with built-in parallelism. Therefore, the parallelism, although not explicitly specified, depends on the use of these skeletons in the program. A very powerful concept for describing parallelism is provided by process control languages. Most closely related to our parallel programming technique discussed in Section 4.3 are Caliban (Kelly 1989) and first-class schedules (Mirani & Hudak 1995). Both systems provide separate control languages that can use functional expressions in specifying a structure of parallel processes. These systems will be discussed in more detail in Section 4.9.1. On the side of explicit parallelism, extensions to Lisp, such as MultiLisp (Halstead, Jr. 1985) and Mul-T (Kranz et al. 1989), have to be mentioned. The basic construct used in these languages, a future, is closely related to the `par` in GpH. Section 5.7.1 discusses this relationship in more detail. Other systems that provide explicit annotations for controlling parallelism are Concurrent Clean (Nöcker, Smetsers, van Eekelen & Plasmeijer 1991), Hope$^+$ (Kewley & Glynn 1989), and the system proposed by Burton (1984).

## 2.3 Implementation of Functional Languages

This section discusses different approaches to the implementation of functional languages. The discussion focuses on the graph reduction and the dataflow models. They present a high level of abstraction and thereby incorporate parallel execution in a very natural way. Hammond (1994) presents a detailed discussion of different models for the parallel implementation of functional languages. Schreiner's annotated bibliography (Schreiner 1993) gives a comprehensive survey of the parallel implementation of functional languages.

Historically, the first implementations of functional languages used a *stack-based approach* such as the SECD machine (Landin 1964), which has been extended to lazy languages by Burge (1975) and Davie & McNally (1990). The SECD-M machine adds concurrent threads and non-determinism to the basic design (Abramski & Sykes 1985). Both the eager and the lazy SECD machine are described in detail in Field & Harrison (1988)[Chapter 10].

Another approach is to use a fixed set of combinators, such as SK combinators known from combinatory logic (Curry & Feys 1958), as the abstract machine language. The implementation of SASL was based on this design (Turner 1979). Later this approach was extended to use program dependent super-combinators (Hughes 1984). A super-combinator is obtained from a function body by lifting maximal free expressions, i.e. the largest sub-expressions which contain free variables. This transformation maintains the full laziness property that no expression will be evaluated twice, and differs in this aspect from the more basic $\lambda$-lifting transformation (Johnsson 1985). The categorical abstract machine (Curien 1986) combines the environment-based approach of the SECD machine, which is defined via state transitions, with the idea of using basic variable-free combinators out of combinatory logic as the abstract machine language.

### 2.3.1 The Graph Reduction Model

The graph reduction model is based on the idea of representing the program as a graph structure and defining evaluation as rewriting this graph (Wadsworth 1971, Peyton Jones 1987). Figure 2.1 shows the lazy evaluation process of the expression `square (1+2*3) where square x = x*x`. Note that in the first step two redexes can be reduced in parallel: the definition of `square` can be applied and the expression `2*3` can be reduced. The latter is possible because `square`, multiplication, and addition are strict. This example also shows how several instances of the parameter `x` are shared when applying `square` to a concrete argument. This avoids duplication of work.

This approach has several advantages:

- It is easy to express sharing of program expressions by sharing in the graph;

- a call-by-need evaluation can be easily implemented by overwriting the reduced node with its result;
- independent parts of the graph can be evaluated in parallel.

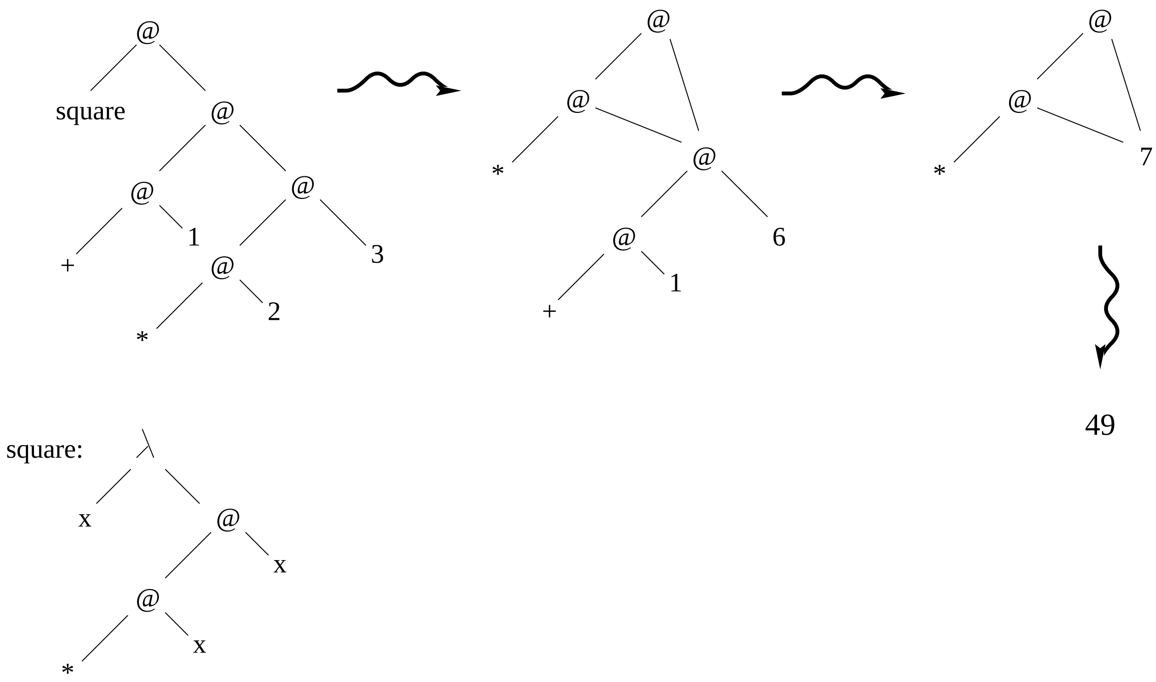


**Figure 2.1** The principle of parallel graph reduction

Because of the first two advantages most modern non-strict languages are implemented using graph-reduction. However, this pure graph reduction model is very high-level, and a straightforward implementation is rather inefficient. For example, the reduction process described in Figure 2.1 suggests an interpretive implementation, solely operating on graph structures. In comparison most modern abstract machines use an approach of compiled graph reduction. Rather than using a top level interpreter, each node in the graph, a "closure", contains code for performing a reduction. In particular, user defined functions are compiled into code that simulates the construction of a graph structure. The generated code typically uses an evaluation stack to perform built-in operations, such as basic arithmetic, more efficiently, without the need to allocate heap objects in this case. The G-machine (Johnsson 1987, Augustsson 1987) was the first machine that used compiled graph reduction, eliminating most of the interpretive overhead in the execution of non-strict LML programs. Many later abstract machines were based on the G-machine, e.g. the Spineless G-machine (Burn et al. 1988), the Spineless Tagless G-machine (Peyton Jones 1992) etc. Peyton Jones (1987)[Chapter 20] gives a good overview of different optimisations of the basic graph reduction mechanism.

From this thesis' point of view the most important advantage of the graph reduction model is the ease of expressing parallel computation in this model. A parallel graph reduction model can be very naturally expressed as a spark pool, i.e. a pool consisting of pointers to unevaluated expressions ("thunks"), and a set of processors that take sparks out of this pool and execute them by creating a *thread*, an independent process performing standard graph reduction. These threads are kept and maintained in a separate thread pool. In our model adding a new spark to a spark pool is performed by a `par` combinator. Mutual exclusion between threads trying to reduce the same piece of graph has to be guaranteed, this will be discussed in Section 2.4.1. Peyton Jones (1989) discusses parallel graph reduction in detail.

## 2.3.2 The Dataflow Model

Another high-level computation model that does not require a sequential evaluation mechanism is the dataflow-model (Dennis 1974). The idea in this model is to represent operations as nodes in a graph and to represent data as tokens passed between the nodes. Evaluation is governed by the "firing rule": a node with tokens on every input arc consumes these tokens, applies its function to their values, and sends a result token with this value to its output arc. In short, the node "fires".

### The Principle of the Dataflow Model

In contrast to the demand-driven graph reduction model, the dataflow model is *data-driven*. The evaluation of operations is determined by the availability of data rather than by the demand on a result. Thus, a natural evaluation mechanism is based on eager evaluation. This aims at exposing a maximal amount of parallelism in the system, even if some of the parallelism is speculative.

It is important to distinguish the operational aspect of the evaluation model from the semantic aspect of strictness. Although parallel eager evaluation is safe in a strict language, e.g. SISAL (Böhm et al. 1991), modern dataflow language such as Id (Nikhil 1989) and pH (Aditya et al. 1995) are non-strict in order to minimise data dependencies in the program. The runtime-system guarantees that the failure of one evaluation does not necessarily result in a failure of the overall computation.

Figure 2.2 demonstrates the execution of the expression `square (1+2*3) where square x = x*x` in the dataflow model. Here the nodes in the graph are operators and the arcs represent data dependencies. The graph is unchanged throughout the computation. In the first step the `*` operator can fire because both arguments are available, whereas the `+` operator has to wait for its second argument. Within the `square` function the result token from the previous computation (7) is duplicated, corresponding to sharing the result of an expression in the dataflow model.

### Optimisations in the Dataflow Model

The dataflow model aims at exposing a maximal amount of parallelism. Historically, it was mainly used as a concrete machine model for special purpose dataflow machines with special hardware support for the basic machine operations, e.g. the Tagged-Token Dataflow Architecture (Arvind & Nikhil 1990), the Manchester Dataflow machine (Gurd et al. 1985), the Monsoon machine (Papadopoulos & Culler 1990), Sigma-1 (Shimada et al. 1986), PIM-D (Ito et al. 1986) etc. More recent abstract dataflow machines significantly depart from the pure dataflow model and use a control-flow language as machine independent intermediate language, e.g. the TAM machine (Culler et al. 1993) and *T(Chiou et al. 1995). However, an implementation on stock hardware still faces serious efficiency problems and to overcome these problems many optimisations to the basic model are performed.

One of the major inefficiencies of the dataflow model is the extremely fine-grained parallelism. Every primitive operation can be implemented as one node in the dataflow graph. This yields a high overhead in the parallel execution of the program. Therefore, special compile time methods for partitioning the dataflow graphs and merging the partitions into "macro dataflow nodes" have been developed (Sarkar & Hennessy 1986). For example, the Id90 compiler for the TAM machine (Culler et al. 1993) iteratively computes dependence and demand sets between nodes in the

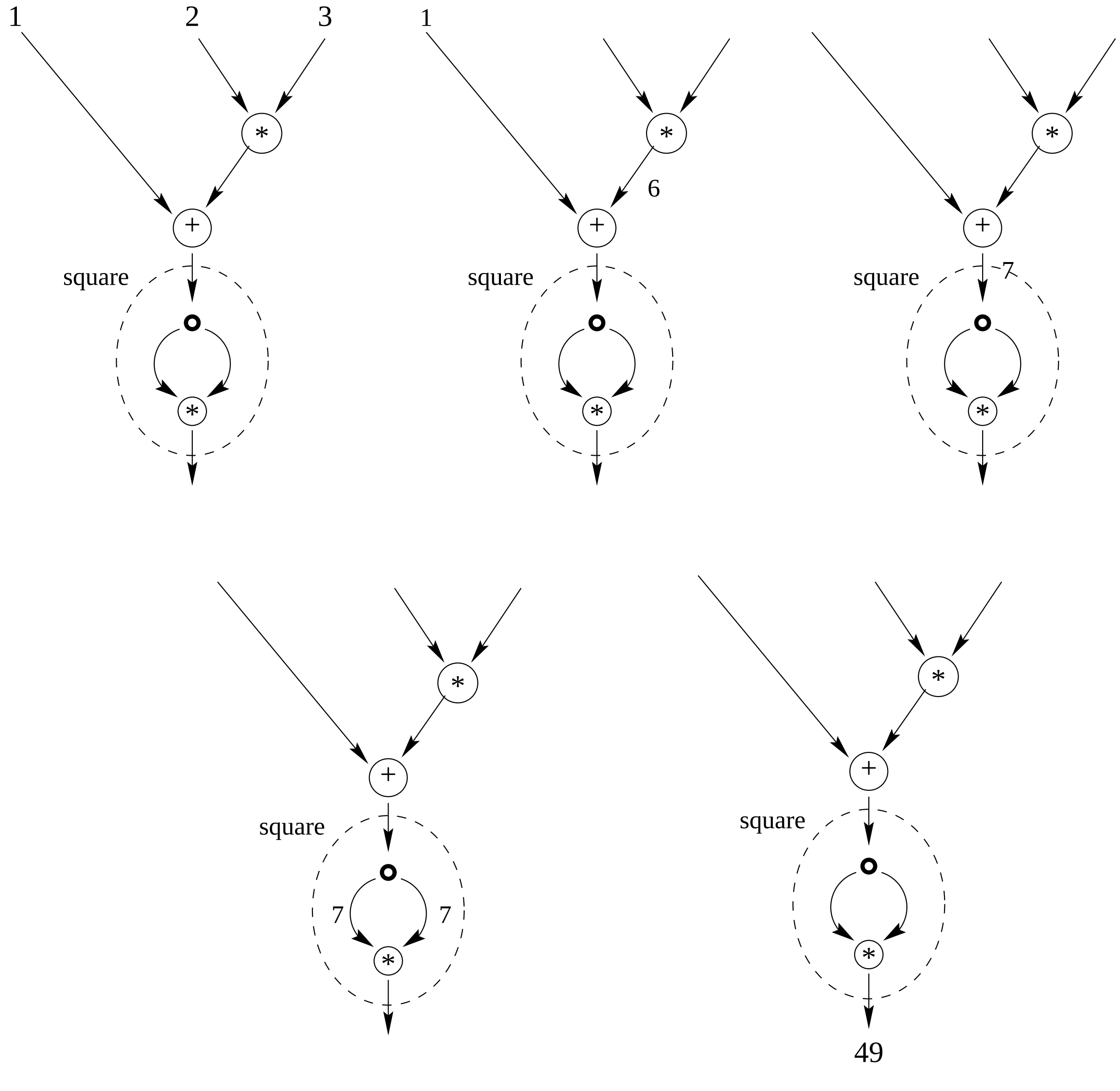


**Figure 2.2** The principle of the dataflow model

dataflow graph merging independent nodes into macro nodes. Each of these nodes is then realised as a thread in the abstract machine. This analysis can also be done globally as is shown in Traub et al. (1992). Furthermore, TAM distinguishes between coarse-grained frames, which are the units of computation and are allocated to processors, and these more fine-grained threads operating within a certain frame.

Having many threads within a frame guarantees latency tolerance in a multi-threaded scheduling environment.

As further optimisations several mechanisms from the compilation of imperative languages are integrated into the dataflow model:

- *Activation Frames:* Machines like Monsoon use bits in an activation frame to indicate availability of a result. In older dataflow architectures an expensive associative store was used.
- *Memory:* Constructs like I-structures (Arvind et al. 1989), single assignment variables, and M-structures (Barth et al. 1991), mutual exclusion variables, are used for storing and retrieving values. In contrast, the pure dataflow model has no store.
- *Split Phase Operations:* Access to I-structures and M-structures is performed via split phase operations, i.e. after executing the operation the thread will be automatically descheduled. This is done to overlap communication with computation via variable access.
- *No explicit dataflow graph:* The latest compilation model for pH avoids the use of dataflow graphs as an intermediate language (Arvind et al. 1996). Instead it uses a sugared version of a call-by-need $\lambda$-calculus, the $\lambda_S$-calculus, with letrec to express sharing, with barriers for explicit synchronisation, and I-/M-structures.

In summary, these optimisations in the dataflow model, as well as the optimisations in the graph reduction model discussed in detail in Section 2.4 show a convergence towards adopting efficient techniques developed for parallel imperative languages.

### 2.3.3 Other Models

Although the SECD, graph reduction and dataflow models are the best known models for parallel functional programming, many other approaches towards a parallel implementation have been suggested. This section discusses some of these models.

The Gamma model (Banâtre & Le Métayer 1990) uses the metaphor of chemical reactions to describe parallel evaluation. In this model an evaluation step resembles the chemical reaction in a pool, a multiset, of atoms: first a matching group of objects in an object pool is selected, then an operation on these objects is performed and these objects are replaced with result objects. This "rewriting" is repeated until no more matching objects can be found. In this model a program is specified by the functions for matching and evaluation. This model facilitates a high level program derivation approach as well as parallel computation because reactions on disjoint sets of atoms can be performed in parallel. However, an implementation faces problems of efficiently matching objects, similar to the problems met in token-based dataflow implementations. Gladitz & Kuchen (1996) describe a parallel implementation of this model on a shared memory multi-processor.

The NESL system (Blelloch 1996) uses a model of nested data parallelism. It is programmed in an SML-like, strict, higher-order language. Parallelism can only be expressed implicitly via using sequence operations, similar to Haskell's list comprehensions, and via higher-order functions that process sequences in a data-parallel fashion. Again this restriction facilitates an efficient implementation of the language. It is mainly used for running numerical algorithms on supercomputers such as CRAY Y-MP, Connection Machine CM-2, and Encore Multimax.

Finally, several models have been designed for the efficient execution of specific parallel programming paradigms. The idea here is to gain improved efficiency for a restricted but important set of programs. One example of such a machine is ZAPP (Burton & Sleep 1981, Goldsmith et al. 1993), which has been designed for the efficient parallel execution of divide-and-conquer programs. It performs parallel computation on a virtual tree of processors. Communication is performed by message passing. No global heap is implemented in this system. Experiments on a transputer based implementation of this machine reported nearly optimal speedups for some divide-and-conquer programs like n-queens (McBurney & Sleep 1987).

## 2.4 Runtime-System Issues

This section discusses key aspects of the runtime-system in a parallel functional language that are crucial to the performance of parallel programs. This discussion will focus on a model of parallel graph reduction. However, most of these aspects are central to any implementation of a parallel functional language.

Many of the issues discussed in this section can be hidden behind a distributed, or virtual shared memory implementation and lightweight threads. This has been done for SISAL (Freeh & Andrews 1995, Haines & Böhm 1992) and in Rushall's implementation of lazy task creation on top of a Spineless G-machine (Rushall 1995). However, in this approach the possibility of directly influencing low-level issues, via the compiler, and optimising the system for a particular computational model like graph reduction are lost. Therefore, such an approach is usually just used for prototyping rather than for optimised parallel machines. This approach will not be discussed in greater detail.

### 2.4.1 Evaluation Models

A major issue in the evaluation model is

*How are the parallel threads created and synchronised?*

In a parallel implementation it can, and probably will, happen that two parallel threads try to evaluate the same expression. The evaluation model specifies

- how and when parallel threads are generated (*sparking*),
- how to prevent the threads from evaluating expressions already under evaluation (*locking*) and
- how to keep track of and ensure data transfer to threads that need the result of an ongoing computation (*waiting list*).

These three issues describe the interaction between parallel threads and the conceptually shared heap. Another issue that is discussed in this section is the synchronisation mechanism between the parallel threads. In particular the following models can be used:

- a notification model,
- a fork-and-join model,
- and an evaluate-and-die model.

**Sparking:** The most commonly used mechanism for generating threads in the graph reduction model is a *sparking* mechanism (Clack & Peyton Jones 1986). This mechanism assumes that all parallelism has been exposed on the abstract machine language level. This can be achieved via annotations either in the source code or at some stage in the intermediate or abstract machine code. When such a parallelism annotation is encountered in the code, a spark, usually a pointer to a thunk, is created (see Page 24 for a discussion of the parallelism annotations).

There are at least two ways to interpret these sparks. They can be either *ignorable*, in which case they represent potential parallelism but the runtime-system is free to discard sparks, e.g. when the load of the machine is too high; or they may be *mandatory*, in which case a thread has to be created for this spark eventually. The latter variant is more sensitive towards fine-grained parallelism whereas a model of ignorable sparks yields a high flexibility in the amount of parallelism that is created, by dynamically combining threads. These benefits of ignorable sparks come for the price of increased overhead in maintaining a pool of available sparks. Ignorable sparks are used in many designs such as GRIP (Peyton Jones et al. 1987), GUM (Trinder, Hammond, Mattson Jr., Partridge & Peyton Jones 1996), $\langle \nu, G \rangle$-machine (Augustsson & Johnsson 1989), PABC machine (Nöcker, Plasmeijer & Smetsers 1991). Some machines like the HDG machine (Kingdon et al. 1991), and the $\nu - STG$-machine (Hwang & Rushall 1992) use both versions of sparks.

Another way of exposing parallelism during the execution of the program is based on the idea of just *seeding* enough information in the runtime stack to allow the extraction of parallelism later on. The motivation of this approach is to further reduce the overhead of managing parallelism in the case of sequential execution. The price that has to be paid is additional overhead for extracting parallelism out of the seeded stack. Rushall (1995) presents an implementation of this idea on top of the Spineless G-machine, implemented on a KSR1 multi-processor. Goldstein et al. (1996) have implemented a similar scheme in the context of the TAM machine, which is based on dataflow inspired compilation. He reports significant runtime improvements for rather large programs on a CM-5. A more detailed discussion of these mechanisms is given in Section 5.7.1.

**Locking:** The standard way to implement synchronisation between threads that try to evaluate the same thunk is via *locking* the node as soon as evaluation starts. If a thread encounters a locked node it joins a *waiting list* attached to the locked node. When the node is updated with the result of the evaluation, all threads in the waiting list have to be reawakened. This is the basic mechanism used in GUM(Trinder, Hammond, Mattson Jr., Partridge & Peyton Jones 1996), the $\langle \nu, G \rangle$-machine (Augustsson & Johnsson 1989), the PABC machine (Nöcker, Plasmeijer & Smetsers 1991), GAML (Maranget 1991), EQUALS (Kaser et al. 1997), in fact in most parallel graph reduction machines.

It is critical for the performance of the parallel machine to have efficient locking of nodes as well as enqueuing and awakening of threads, because evaluating a node and updating it with its result are very common operations in a graph reduction system. Therefore, many optimisations to this basic scheme have been studied.

For example, locking a node may be a rather expensive operation requiring atomicity. To reduce these costs the GAML system distinguishes on language level between application nodes that might be shared and those that are known not to be shared. No locking is required for the latter. In general a sharing analysis, e.g. (Jones & Le Métayer 1989), would be useful to determine whether a node may be shared. If the intermediate language uses a special `letpar` construct for binding expressions that may be evaluated in parallel, locking is only necessary for such `letpar`-bound variables (Hogen & Loogen 1994). However, it is unclear whether this optimisation is desirable in all cases. For example the STG-machine uses a locking mechanism, "black holing", even in a sequential setup. This has two important advantages: a cycle in the program can be easily detected because the enter code of a black hole produces an error message, and by overwriting the thunk with a black hole heap space for the arguments can be freed before the thunk is updated, which helps to avoid space leaks. Giving up these advantages is probably only reasonable for an optimising compilation.

In order to implement locking efficiently, some machines like the $\langle \nu, G \rangle$-machine, the HDG machine, the EQUALS, and the GAML system use a bit in the node to mark it as being under evaluation. Other machines like the GUM or the PABC machine, which are based on a tagless design, change the code pointer of the node such that entering the node causes the thread to be suspended and added to the waiting list. This approach saves a test operation on entering a node.

**Waiting List:** In order to record, which threads are waiting for the result of a computation, waiting lists are usually used. In the graph reduction model, where the result overwrites the original node, the waiting list is usually attached to the locked node. This mechanism makes use of the fact that the descriptors for threads are heap allocated and can be referred to by closures without any modification to the evaluation model.

All stages from locking, enqueuing a competing thread into the waiting list, updating, and reawakening the thread are depicted in Figure 2.3. In this case thread `A` starts evaluating the depicted graph structure and locks the root closure upon entry. When thread `B` tries to access the root it finds the closure locked and `B` is added to the, so far empty, waiting list of the root closure. Finally, `A` finishes evaluating the graph and updates the root closure with the result. Upon updating the waiting list, containing `B`, is reawakened and `B` can continue with its evaluation.

In order to minimise the heap usage of the program many abstract machines reuse parts of the node for the root of the waiting list: GUM uses the first two words of the closure, the $\langle \nu, G \rangle$-machine uses the back-link in the graph structure. The key observation, which allows such reuse of parts of a node, is that a waiting list will only exist when the node is locked. In this case, only two operations can be performed on the node: adding a thread to the waiting list and updating the node with the result. In both cases, no direct access to the data stored in the closure is necessary.

The PABC machine reserves space for the root of a waiting list in every node. This reduces the overhead of locking a node but increases the heap usage. However, the optimisation of the PABC machine described in Kesseler's transputer implementation (Kesseler 1996) also stores the root of the waiting list in the argument fields of the locked node.

An simpler alternative to using a waiting list is *polling*: a thread that reaches a node under evaluation is not removed from the list of runnable threads and it tests whether the node has been overwritten to normal form whenever it is rescheduled. This eliminates the waiting list overhead but imposes a high load in the presence of fine-grained parallelism. A polling mechanism has been implemented and assessed in the Concurrent Clean system (van Groningen 1992). The results show that even with optimisations to this basic mechanism it is more expensive than a waiting list mechanism if the program is fine-grained.

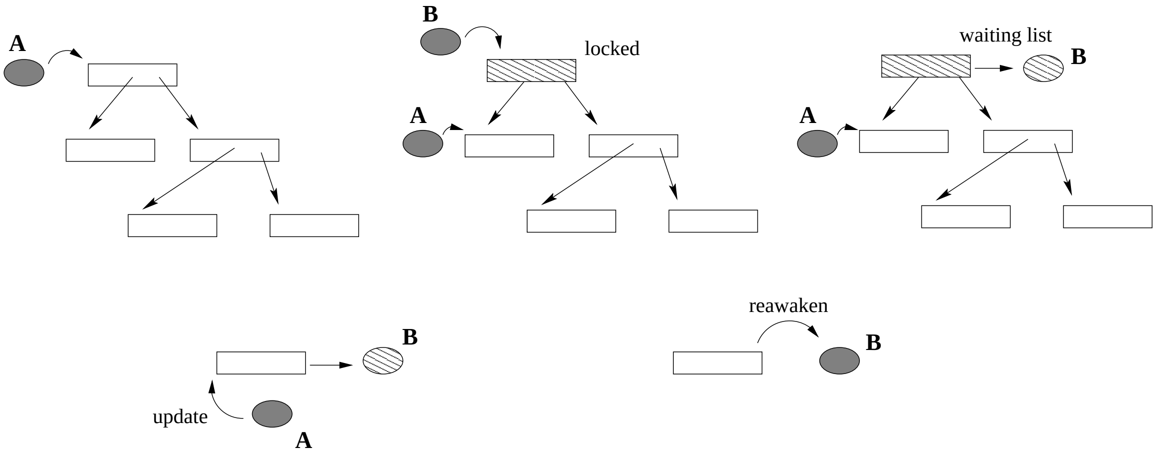


**Figure 2.3** Locking of closures and generation of waiting lists

The pure dataflow model achieves synchronisation between processes by passing tokens. However, on conventional hardware such an approach has proven to be too inefficient. Instead, *I-structures* (Arvind et al. 1989) are commonly used as the central means of synchronisation between threads. The behaviour of I-structures is very similar to those of waiting lists. Initially, these single-assignment variables are empty and a read access is deferred. Since all memory access operations are split-phase operations, a deferred read causes an implicit suspension of the reading thread. A list of deferred read requests has to be maintained for each I-structure cell. When a value is written into the I-structure the read requests can be satisfied by sending messages to the requesting processes. An arrival of such a message will reawaken the suspended process. This mechanism of synchronisation is used in the Monsoon (Papadopoulos & Culler 1990) and *T architectures (Chiou et al. 1995), in the TAM machine (Culler et al. 1993) and in the pHluid system (Flanagan & Nikhil 1996).

The evaluation model of Alfalfa (Goldberg 1988*b*) is one of heterogeneous graph reduction. In general, this is realised via standard locking of nodes and enqueuing of tasks as described above, but all sparks are mandatory. However, in order to optimise the execution of sequential components within the program, a stack-based execution model is provided, too. The stack-based model does not have to deal with parallelism issues because each thread performs sequential execution without being interrupted.

A distinction between such sequential threads and general parallel threads is made in the intermediate language. This uses information automatically inferred by the compiler.

### The Notification Model

In the notification model every child thread is required to notify its parent thread upon finishing its computation. The parent thread is blocked until all of its children have finished. Usually, this is implemented via a pending counter and an associated pending list, the same as a waiting list, of all threads that need the result of this evaluation.

One of the first machines that used such a kind of synchronisation mechanism was ALICE (Darlington & Reeve 1981), which influenced the design of many later machines such as Flagship (Keane 1994), which uses a data-driven rather than a demand-driven model, PAM (Loogen et al. 1989), the HDG machine (Kingdon et al. 1991) etc. These more recent machines use compiled rather than interpreted graph reduction, thereby gaining far higher sequential performance.

The larger-grain graph reduction model (LAGER) (Watson 1988) uses a notification model of synchronisation between parallel threads. However, this model uses seeding rather than sparking in order to expose parallelism. By default, the code is executed in a sequential manner, in order to use optimised sequential code most of the time. At statically determined points, code for generating parallel threads is planted.

The evaluation model in the dataflow-oriented TAM machine (Culler et al. 1993) also uses explicit synchronisation counters, similar to pending counters, for synchronisation across threads. In TAM a thread is a linear sequence of instructions without branching or creating parallelism, somewhat similar to a basic block. A hierarchy of controlflow units is defined, from fine-grained, cheap operations, e.g. inlets for handling messages, to coarse-grained operations with a comparatively expensive synchronisation mechanism. An important difference to the notification model is the fact that synchronisation is performed via data-structures, as in the evaluate-and-die model (see Section 2.4.1), rather than directly between threads.

### The Fork-and-Join Model

The fork-and-join model is a special version of the notification model, which implies symmetric parallelism. A thread that creates other threads becomes a parent process waiting for the results of the children. Thus, synchronisation is performed directly between threads. The fork-and-join model generates a strict hierarchy of threads where a parent can only continue after all children have completed. This restriction allows to use efficient mechanisms for load balancing. However, a fundamental problem of this model is that the usually small computation in the join phase tends to form a parallelism bottleneck.

The Dutch Parallel Reduction Machine (DPRM) (Barendregt et al. 1987) uses such a fork-and-join model. A special "sandwich" annotation has to be used to generate child threads. This annotation has been designed for divide-and-conquer parallelism: it specifies a list of sub-computations that should be done in parallel, and a combination function. The characteristic feature of this annotation is the reduction of all arguments of the sub-computations to normal form before generating parallelism. This avoids bottlenecks of sharing data structures between different threads because data in normal form can be safely copied. It is up to the programmer to use this annotation on expressions of appropriate size in order to generate coarse granularity. However, special mechanisms are necessary to improve the granularity in particular to avoid harmful thread migration in the join phase (Hofman et al. 1992).

### The Evaluate-and-Die Model

In contrast to the previous models, the evaluate-and-die model (Peyton Jones et al. 1987) generates asymmetric parallelism. A thread that creates (potential) parallelism does not have to synchronise with the generated child thread, i.e. it forgets about all generated work. The only means of synchronisation is via the graph structure the threads are working on. In particular, if a thread requires the result of a potentially parallel sub-expression, it will start to evaluate that expression itself, thereby *subsuming* the computation of another spark. In contrast, the notification model would cause the thread to block on the thread evaluating the sub-expression. In the case of a high load, i.e. many runnable threads, such subsumption of sparks automatically increases the granularity of the threads and reduces the number of parallel threads that are generated. However, this mechanism only works for certain, hierarchic structures of computation such as divide-and-conquer. In his thesis Roe (1991)[Section 6.5] shows that evaluate-and-die cannot improve the granularity for some data-parallel programs, which typically exhibit a flat structure of sparks.

GUM(Trinder, Hammond, Mattson Jr., Partridge & Peyton Jones 1996) uses an evaluate-and-die model with ignorable sparks and waiting lists. The $\langle \nu, G \rangle$-machine (Augustsson & Johnsson 1989) uses similar techniques, however, it has been designed for shared memory systems and therefore it splits the heap in chunks to be more flexible in managing the heap sizes of the individual processors. The HDG machine (Kingdon et al. 1991) uses an evaluate-and-die model with tags in each closure indicating whether a task for evaluating this closure has been created and whether the evaluation of the expression has already begun. It uses both ignorable and mandatory sparks assigning them different priorities in a transputer based implementation.

### 2.4.2 Storage Management Models

In a general model of distributed memory an important question is:

> *How is the heap distributed between processors?*

One possibility to model the distributed nature of the heap in a parallel system is to add a new type of closure: a *FetchMe*, or global indirection, closure. It points to a graph structure on a remote processor. When a thread tries to evaluate a FetchMe closure, a fetch request for this graph structure is sent to the remote processor. The thread gets blocked on the FetchMe closure and will be reawakened upon arrival of the graph structure. The same mechanism as for blocking on a closure under evaluation can be used in this scheme. If the remote graph structure is itself under evaluation the fetch request will block on the locked closure. The reply will be sent only after having evaluated the graph structure. This means that the perceived latency in the system is unbounded as it depends on the computations being performed on other processors. It is therefore important to provide latency hiding mechanisms that allow to overlap the communication with useful computation.

The unbounded perceived latency also underlines the importance of *data locality* in order to avoid communication. By data locality we mean the distance between data structures required within one thread of computation, where the unit of distance is one processor. The goal is to keep all required data on the same processor, avoiding communication and thereby improving parallel performance. In sequential implementations a stack ensures data-locality and the efficient use of storage. The importance of using a stack-oriented evaluation in order to maintain data-locality has already been shown in the implementation of heterogeneous graph reduction (Goldberg 1988*a*, Goldberg 1988*b*) for lazy functional languages. In Goldberg's model a stack-based model is used in the sequential parts of the computation in order to achieve high sequential performance and only for the parallel components a packet based model of graph reduction is used.

In a parallel system conceptually each thread needs its own stack. Because the creation structure of threads is a tree the stack becomes a *cactus stack*, with thread creation causing a new branch in this stack. The portion of the stack generated before thread creation is shared between child and parent thread. In subsequent sections the following possible implementations of a cactus stack are discussed:

1. a linked list of *packets*;
2. a linked list of *stack segments*;
3. a *contiguous stack* that is reallocated when needed; or
4. a *meshed stack*.

An area related to the storage management model in a parallel system is parallel, or more general distributed, garbage collection. However, it is not directly relevant to the issues studied in this thesis and will not be discussed in detail. Plainfossé & Shapiro (1995) give an excellent survey of distributed garbage collection techniques.

#### Packet-based Models

The first designs of parallel graph reduction machines, such as ALICE (Darlington & Reeve 1981, Harrison & Reeve 1986), used a packet-based reduction method: conceptually variable size packets are used to hold the arguments to the code as well as local variables needed during the execution of the code. These packets, or frames, are linked together during runtime thereby creating a cactus stack structure with each packet playing the role of an activation frame. Such a packet-based model does not have an runtime allocation overhead because the allocation is done at compile time when generating the closure. However, it uses much more heap space and the danger of space leaks is much higher because if a closure is still live so are all local variables in its frame. In essence, some of the allocation overhead has been moved to the garbage collector.

The HDG machine (Kingdon et al. 1991) uses such a packet-oriented approach. It uses a special "stacklessness analysis" (Lester 1989) to determine the size of the activation record needed to evaluate a node. With this information it is possible to allocate all the required stack space in the node itself. No explicit checks for stack overflow are required at runtime. In contrast, the $\langle \nu, G \rangle$-machine (Augustsson & Johnsson 1989) and the PAM machine (Loogen et al. 1989) may have to extend the space allocated for one packet if a generic function application, which does not contain information about the arity of the function, is used.

### Segmented Stack Model

In this model the stack is allocated in the heap but separated from closures in the graph. By splitting the stack into segments this model can efficiently handle small threads without wasting space on a large stack. For large threads it must be possible for the stack to grow by allocating new segments. In contrast to the packet-based model, these segments are separate from activation frames and changing the size of the stack segments can be a useful tool in the performance tuning stage of parallel program development.

Of course, the increased flexibility imposes some runtime overhead when allocating new stack segments. However, in practice segment sizes are chosen high enough to avoid the creation of long lists of stack segments even if this leads to some waste in heap space. One particular danger of this model is "stack thrashing": if the stack grows and shrinks rapidly across segment boundaries many segments have to be allocated. Additionally, to increasing the runtime overhead this creates a lot of garbage stack segments, which increases the garbage collection rate, unless garbage stack segments are kept on a special list for further reuse. Therefore, it might be better to leave some headroom in each stack segment that can be used upon returning from a discarded stack segment. The GRIP (Peyton Jones et al. 1987) and GUM (Trinder, Hammond, Mattson Jr., Partridge & Peyton Jones 1996) machines use such a segmented stack model.

### Contiguous Stack Model

In this model each thread uses a monolithic block of heap space as its own stack. This achieves good data locality, similar to the sequential evaluation. If the stack runs out of space it has to be enlarged using standard re-allocation. However, this is a rather expensive operation and should be avoided whenever possible. Therefore, in practice rather big stacks are used.

The main disadvantage of this scheme is the huge waste of heap space if many threads do not require a lot of stack. This model is far less suited to dealing with threads of different sizes compared to the previous two approaches. The PABC machine (Nöcker, Plasmeijer & Smetsers 1991) uses this kind of stack model.

### Meshed Stack Models

The meshed stack technique eliminates a parallelism overhead in case of sequential computation by interleaving all local stacks into a single stack. This avoids the necessity of allocating the stack in the heap. This concept was first introduced under the name of spaghetti stack by Bobrow & Wegbreit (1973). The main idea is to mark activation frames that are not on top of the stack as garbage and to run a special compacting garbage collector on the meshed stack if it runs out of space.

This mechanism drastically reduces the overhead when sequential execution is performed because there is no need for allocating new stack segments. It also achieves very good data locality because data is not attached to closures in the heap. However, since the single meshed stack is a centralised resource, it is very hard to implement thread migration on top of this stack model. The meshed stack model has been introduced for the PASTEL machine (Hogen & Loogen 1994) and was inspired by the handling of backtracking in the Warren Abstract Machine (WAM) for implementing logic languages (Warren 1983). Measurements comparing this model with a packet-based model using an interpreter on a transputer system show that the amount of heap allocations is reduced up to a factor of two and the runtime improves by about 20% (Hogen & Loogen 1995).

### 2.4.3 Communication Models

This section tackles the following question:

*How is data exchanged between processors?*

One of the main sources of overhead in a parallel system is communication. In most parallel architectures communication is much more expensive than computation. Therefore, it is very important to provide good data locality in order to avoid communication.

To this end, it is useful to distinguish several aspects of the communication model:

1. *Data placement:* Is data moved to a thread or vice versa?
2. *Latency hiding:* Can the communication be overlapped with useful computation?
3. *Packing:* How much data should be sent in one packet?

An important issue for hiding communication costs is *multi-threading*, i.e. a scheduling method that allows the interleaved execution of several threads of computation. In particular, it is possible to deschedule a thread waiting for data and to schedule another thread, which can perform computation in the meantime. This section discusses details of this method.

#### Data Placement

One important issue for the data locality in the system is data placement, which describes how to handle the distribution of data during the execution of the program. Whenever the result of a remote thunk is required by a local thread there are two possibilities of communication:

- Send the thunk to the demanding process, evaluate it locally by this process and replace it with a global indirection on the remote processor (*local evaluation*).

- Start a thread on the remote processor to evaluate the thunk and then send the result to the demanding process (*remote evaluation*).

The advantage of the local evaluation scheme is that it minimises the delay in obtaining the result. Furthermore, parallelism is only created via picking a spark from a spark pool, not as a side effect from receiving a message. This simplifies load management. The local evaluation model is used in Flagship (Keane 1994), GRIP (Peyton Jones et al. 1987), and GUM (Trinder, Hammond, Mattson Jr., Partridge & Peyton Jones 1996). The remote evaluation scheme, however, might increase data locality by avoiding a distribution of subgraph structures. Because of the potentially high hidden latency imposed by performing the evaluation on the remote processor, an effective latency hiding mechanism is required. There is a higher danger of a severe load imbalance attached to this scheme if no thread migration is provided because some processors may become hot-spots of computation. The remote evaluation scheme is used for example in the PABC machine (Nöcker, Plasmeijer & Smetsers 1991, Kesseler 1996), in the proposed $\nu - STG$-machine (Hwang & Rushall 1992), in PAM (Loogen et al. 1989), and in the related PASTEL (Hogen & Loogen 1994) machine. Alfalfa combines the remote evaluation scheme with an active work distribution scheme which sends available work to idle processors, rather than have idle processors ask for work.

### Latency Hiding

The *latency* in a parallel machine is the time required to send one piece of data between two processors. In practice, latency often varies between pairs of processors and also depends on the network traffic. One way of reducing the impact of the communication costs on the performance of the system is *latency hiding*. The idea of this scheme is to overlap the communication with some useful computation on the local processor. In general, when a thread requests remote data the processor can either:

- block while waiting for the data (*synchronous communication*) or
- execute another thread (*asynchronous communication*).

The second option imposes some overhead on the runtime-system because it has to support multi-threaded scheduling on each processor. However, as a result it is possible to hide the latency in the system if at every point when data is requested enough parallelism is available to perform useful computation.

In a model of *synchronous communication* a processor is blocked if a thread requests remote data. This kind of communication only makes sense if the ratio of latency to the time needed for scheduling is very small. In such a case it is more efficient for the processor to block on a thread that is waiting for remote data, rather than deschedule it and look for another thread to run.

In contrast, *asynchronous communication*, allows other threads to run while one thread waits for the arrival of remote data. This behaviour allows the overlapping of communication and computation and is essential for latency hiding. It is worth noting, that machines based on the dataflow model, which usually generate a huge number of fine-grained threads, put a specific emphasis on latency hiding, e.g. TAM (Culler et al. 1993), *T (Chiou et al. 1995), pHluid (Flanagan & Nikhil 1996). In these models certain instructions like accessing an I-structure or writing to it, cause an automatic descheduling of the current thread. Therefore, these split-phase instructions implicitly define the length of one sequential thread of computation.

### Packing

Finally, the aspect of *packing* has to be considered. The question here is how much data to pack into one packet when transferring data. By developing a pre-fetching packing scheme a graph reduction system can realise a caching scheme that exploits the structural information of the program, which is encoded in the graph. The goal of such a scheme is to reduce the total communication cost by increasing the granularity of the communication. However, if the packing scheme also pre-fetches thunks, which represent work, it may lead to a very uneven load balance and even deteriorate data locality.

In the context of the PABC machine (Kesseler 1996) examines different "copying strategies" for the Concurrent Clean system on a transputer network. Finally, he develops a lazy normal formal copying strategy, which copies normal form closures and only those non normal form closures that are specially annotated in the program. We have implemented several "packing schemes" in the GranSim simulator. In measurements of these schemes, a scheme that packs a full-subgraph generally performed best. However, for some communication-intensive programs a scheme that only packs normal forms performed better. These packing schemes are discussed in detail in Section 3.3.1.

### 2.4.4 Load Distribution

The question that is examined in this section is dual to the question how the heap is distributed over all processors:

> *How is work distributed and balanced between processors?*

From a global point of view it is useful to distinguish two approaches toward load distribution:

- *Passive load distribution* where idle processors have to explicitly ask for work, and
- *active load distribution* where new threads are sent to remote processors.

Passive load distribution, which is sometimes called *work stealing*, tries to minimise the overhead during periods in which all processors are busy anyway. However, this may yield an uneven load distribution if few threads are creating a lot of parallelism. In contrast, active load distribution sends, by default, a new thread to a remote processor for execution. Although this gives a more even load distribution it may yield a deterioration in the data locality of the system. In both cases, however, it is desirable to have load information about other processors available. Obtaining such information may require significant communication and therefore all machines have to find a compromise between the competing goals of an even load distribution and a minimal amount of communication. As a result, many implementations use a random allocation mechanism, e.g. ALICE (Harrison & Reeve 1986).

For example, GUM (Trinder, Hammond, Mattson Jr., Partridge & Peyton Jones 1996), a system of passive load distribution, uses a "fishing" mechanism, where requests are sent to random processors. Some delay is added to avoid flooding the system with work requests, a problem observed on ALICE (Harrison & Reeve 1986), and allowing just one outstanding fish per processor. Because GUM packs more than one thunk into a packet, some pre-fetching of work is performed. The HDG machine

(Kingdon et al. 1991) sends requests only to neighbouring processors. They return work if they have at least two tasks where one of them has not been started, yet. This is similar to the strategy used in ZAPP (McBurney & Sleep 1987). On ZAPP and GRIP experiments have been performed with pre-fetching work, i.e. asking for work when the local pool of work falls below a certain threshold. However, this did not in general yield better performance. The PAM (Loogen et al. 1989) system regularly exchanges information about the workload of neighbouring processors in order to improve the load balance. It uses passive load distribution and exploits the load information in order to decide which processor to ask for work.

In contrast, the Alfalfa machine is based on active load distribution. Extensive studies of various different load balancing schemes (Goldberg 1988*b*) have been performed on this distributed memory architecture. As a result, *diffusion scheduling* with a simple load balancing heuristic performed best. The idea of diffusion scheduling is to send work only to neighbouring processors and to pick the least loaded processor. Thus, only load informations from the neighbours is required. However, this method may react rather slowly to rapidly changing load situation and to hot-spots in the system. On Alfalfa it showed satisfying results, even though no task migration is supported in this implementation.

Issues closely related to load balancing are *load bounding* and *throttling*, which aim at avoiding an excessive amount of parallelism in the system. It is important not to prohibit a large amount of parallelism by design because this would diminish its scalability. However, typically functional languages exhibit an abundance of parallelism, which requires some techniques aiming at limiting the total number of generated threads. Problems with load bounding have been observed on ALICE (Harrison & Reeve 1986), ZAPP (McBurney & Sleep 1987), on PAM (Loogen et al. 1989), and on many dataflow machines. This problem is related to the fine granularity of the threads that are normally created.

A simple but quite effective mechanism for load bounding has been developed on ZAPP (McBurney & Sleep 1987): when the load of the machine is low the runnable queue is treated as a FIFO queue, favouring threads near the root of the divide-and-conquer tree. However, when the load drops below a certain threshold a LIFO mechanism is used. A similar mechanism has been adapted on the Manchester Dataflow machine (Gurd et al. 1985), where a hardware throttle examines the length of the token queue to decide whether a new thread should be generated or whether it should be suspended. In the latter cases it may be reactivated at some later time when the load drops below the threshold (Ruggiero & Sargeant 1987). Similar techniques have been used in the LAGER model (Watson 1988), in the STAR:DUST machine (Ostheimer 1991), and in the $\pi$-RED$^+$-machine (Bülck et al. 1994) via a limited supply of tickets.

## 2.5 Our Model

This section locates the model of GranSim and GUM in the design space outlined in the previous sections. The detailed discussion of GranSim in the following chapter will show that both models are almost identical.

In short, the characteristics of the GranSim/GUM model can be specified as follows:

- Implementation model: parallel graph reduction
- Evaluation model: evaluate-and-die
- Thread placement: local evaluation
- Communication: message passing
- Storage management: segmented stack
- Load distribution: passive
- Scheduling: multi-threading, unfair

The choice of this particular model has been motivated by experiences from parallel functional programming on the GRIP machine (Hammond & Peyton Jones 1992, Hammond et al. 1994), which uses parallel graph reduction, an evaluate-and-die model of computation and passive load distribution. In order to support general parallel architectures message-passing is used for communication. In order to support higher latency systems multi-threading has been added as a means of hiding latency.

**Implementation model:** The implementation model is an extension of the Spineless Tagless G-machine (Peyton Jones 1992). In the parallel system new types of closures such as FetchMe closures (see Section 2.4.2) and waiting lists (see Section 2.4.1) have to be added. Furthermore, the notion of a global address has to be introduced to uniquely identify closures on different processors. Threads and stacks are modelled as special closures.

**Evaluation model:** Our model uses an evaluate-and-die model as described in Section 2.4.1. This model was very successful on GRIP. One of its most important features is the possibility to dynamically increase the granularity of threads. An explicit, distributed spark pool is used for maintaining sparks. One difference between GUM and GranSim is that the latter can use an infinite spark pool.

**Data placement:** In our model we use local evaluation of data that is needed by a thread. In this approach the delay in obtaining a result does not dependent on the load of a remote processor. Therefore, the perceived latency is reduced. In general, however, it is not clear whether local or remote evaluation will yield better results. It is an interesting topic for future work.

**Communication:** The communication is modelled via message passing between different processors. This yields a very portable implementation. By using packing routines that are tailored to graph reduction it is possible to exploit the information contained in the structure of a graph to be sent.

**Storage management:** Our model uses a segmented stack storage management model. This minimises the waste due to too large stacks, and increases the data locality compared to packet based approaches. GUM uses a weighted reference counting mechanism for performing distributed garbage collection (Bevan 1987). However, we will not explore issues related to garbage collection in more detail here.

**Load distribution:** GUM uses a passive model of work distribution by implementing a work stealing mechanism. This mechanism tries to minimise the number of messages required for load distribution, but may produce a rather uneven load balance. GranSim simulates this model but offers more flexibility, for example allowing several steal requests per processor at the same time. No load information is exchanged between the processors.

**Scheduling:** Our model uses multi-threaded scheduling, which is essential to hide latency. In GRIP (Peyton Jones et al. 1987) synchronous communication was used and therefore multi-threading was not necessary.

## 2.6 Summary

This chapter has shown that parallel graph reduction is a very natural model for expressing parallel execution. In common parallel machines its rather high level description of computation requires an efficient mapping of basic operations like locking closures and handling waiting lists onto standard hardware. This is a similar situation as in the dataflow community where the current trend is to depart significantly from the core model, using a few selected standard synchronisation constructs to implement a functional language. One main source of cross-fertilisation in this area has been in adopting an aggressive multi-threading approach within a graph reduction framework.

Two aspects of the dynamic behaviour in a parallel graph reduction system require special attention: data locality and granularity. The former is crucial to avoid unnecessary communication, the latter is essential for minimising the overhead for parallel computation. Chapter 5 will focus on mechanisms for improving granularity. However, before focusing on the issue of granularity the following chapter will describe the underlying parallel machine and its simulator, GranSim, in more detail. In doing so, variants in implementing crucial runtime-system operations, as outlined in this chapter, will be discussed.

# Chapter 3

# GranSim— A Simulator for Parallel Haskell

**Capsule**

The main motivation for simulating the parallel execution of a functional program is to abstract from machine specific details and from the often non-deterministic behaviour of a complex parallel system. Such an abstraction enables the programmer to focus on the parallelism inherent in an algorithm, taking an *algorithm-oriented* view of parallel execution. In order to support such a view a very simple simulator is sufficient. For example, communication costs are often ignored in order to expose the maximal amount of parallelism in the program. The GranSim-Light setup of the simulator presented in this chapter supports this view by modelling an idealised machine with zero communication costs and an infinite number of processors.

For the subsequent studies on granularity, however, such an approach is not sufficient. For studies on this level of detail, involving aspects of the underlying runtime-system, a more detailed *system-oriented* view of parallel execution is taken. For this approach it is crucial to accurately model a wide range of parallel machines that differ in the implementation of basic operations like inter-processor communication. Therefore, flexibility and accuracy are two equally important, though competing, aspects in the design of GranSim. For the overall accuracy of the simulation it is important to achieve a balance between the accuracy of the compilation (to avoid naive generation of inefficient sequential code), of the computation, and of the communication during the simulation. In order to meet these requirements of flexibility and accuracy GranSim has the following crucial features:

- It offers different variants for many basic runtime-system operations like communication.
- It uses a state-of-the-art optimising compiler (GHC) for generating graph reduction code.
- It measures computation time in machine cycles rather than reduction steps.
- It accurately models the communication in a parallel system.
- It offers granularity improvement mechanisms to improve the performance of parallel programs.

GranSim has been used in the parallelisation of several large programs. In this process, it has proven to be robust and to be an important component of the parallel engineering environment. This is being underlined by its current use at several universities worldwide.

## 3.1 Introduction

In the parallel functional programming community simulators are very popular, e.g. (Runciman & Wakeling 1993, Roe 1991, Deschner 1989, Joy & Axford 1992). They allow the programmer to take a very abstract view of parallelism, matching the rather abstract view of computation that is supported especially by lazy functional languages, where definition is cleanly separated from control. However, when running the program on a real machine low-level details of the execution can no longer be ignored. These details may very well be the reason for not obtaining the parallelism that is present on a more abstract level. At this stage the development of a parallel algorithm or the parallelisation of an existing algorithm turns into the performance tuning for a specific parallel machine. Although simulators for exactly modelling such machine details exist (Bennett 1993, Hofman 1994, van Groningen 1992, Keller & Lin 1984, Morais 1986, Watson 1989), they usually lack the ability to model a wide range of parallel architectures.

GranSim, a simulator for the parallel execution of *Glasgow Parallel Haskell (*GpH*)* (see Section 2.2.3), helps the programmer in both stages. Different setups of the simulator reflect different views of the parallel execution: an algorithm-oriented view is supported by the GranSim-Light setup, whereas a less abstract system-oriented view is supported by the standard setup of GranSim. In the latter setup GranSim can simulate most MIMD machines by tuning the available parameters specifying the characteristics of the parallel machine. The limits of such a simulation are discussed in Section 3.5.

GranSim uses a parallel graph reduction model of computation as discussed in Section 2.3.1. The particular implementation is based on the Spineless Tagless G-machine (STGM) (Peyton Jones 1992), with the parallelism annotations `par` and `seq`, which have been discussed in Section 2.2.3. The STGM has been chosen as the underlying abstract machine, because it is used in the Glasgow Haskell Compiler (GHC). Therefore, GranSim can make use of GHC for performing the compilation. The GUM system, a portable parallel runtime-system for Haskell (Trinder, Hammond, Mattson Jr., Partridge & Peyton Jones 1996), uses the same abstract machine and a subset of the same annotations as GranSim. For realising the communication between the processors, GUM uses the PVM communication harness. Thereby, GUM achieves a high level of portability and it has been used on shared-memory machines, distributed-memory machines and workstation networks already. The development of GranSim and GUM was independent, but in several cases influential. As a result of using the same abstract machine, GranSim can be parameterised to closely resemble the GUM system. However, as will become clear from this chapter, GranSim is much more flexible than just simulating the GUM system.

The two main topics studied in this thesis are large-scale parallel programming and granularity. The first topic requires an algorithm-oriented view in developing and tuning a parallel program. A more detailed system-oriented view is needed in order to run it on a particular parallel machine. The study of granularity also requires a system-oriented view in order to model and study different runtime-system features. In particular for the latter view the flexibility and the accuracy of the simulator are of special interest. These issues will be emphasised in the following discussion.

The core system of GranSim has been developed jointly with Dr. Kevin Hammond and Dr. Andrew Partridge. This initial version includes the basic design of the distributed heap, of spark pools, and of thread pools. This design was based on the runtime-system of GRIP for PVM (Hammond 1993) and GRAPH for PVM (Loidl & Hammond 1994), two versions of a port of the GRIP runtime system using PVM to perform communication. The latter added multi-threading and asynchronous communication to the original GRIP runtime-system. Part of the support for multi-threading in GranSim is based on the existing implementation of the GHC runtime-system for Concurrent Haskell (Peyton Jones et al. 1996). The extensions developed in this thesis on top of the core version of GranSim include the design and extension of the communications system with asynchronous communication, several variants of rescheduling, bulk fetching with several variants of packing graph structures (see Section 3.3.1). An extension of the work request mechanism, several granularity improvement mechanisms, and the idealised GranSim-Light setup (see Section 3.4) have been implemented. These extensions are necessary to study a variety of architectures and to specifically focus on granularity aspects of the parallel execution. Finally, GranSim has been integrated into GHC and is now publicly available from the GHC web page (GranSim 1998) for both Haskell 1.2 and 1.4.

The structure of this chapter is as follows. Section 3.2 presents the global structure of the simulator. Section 3.3 discusses its main characteristics, distinguishing it from other simulators. Section 3.4 focuses on the GranSim-Light setup. Section 3.5 addresses shortcomings of the current version of the simulator. Section 3.6 validates the results obtained from GranSim by comparing them with results from HBCPP, GRIP and GUM. Finally, Section 3.7 summarises.

## 3.2 Structure of GranSim

Figure 3.1 shows the global structure of GranSim. In the standard setup GranSim simulates a finite number of processors. The GranSim-Light setup drops this restriction in order to provide an algorithm-oriented view of computation that exposes the total amount of parallelism available in a program. GranSim-Light is discussed in more detail in Section 3.4.

Each of the simulated processors has its own spark pool and thread pool as well as its own clock. Clock synchronisation is performed via accessing the global event queue, which is sorted by the time stamps of the entries in this queue. The spark and thread pools are physically distributed but logically shared. Explicit messages between processors have to be simulated in order to transfer sparks and threads between processors.

The simulation is event driven with events representing actions related to the parallel nature of the program execution like thread creation, communication etc. The

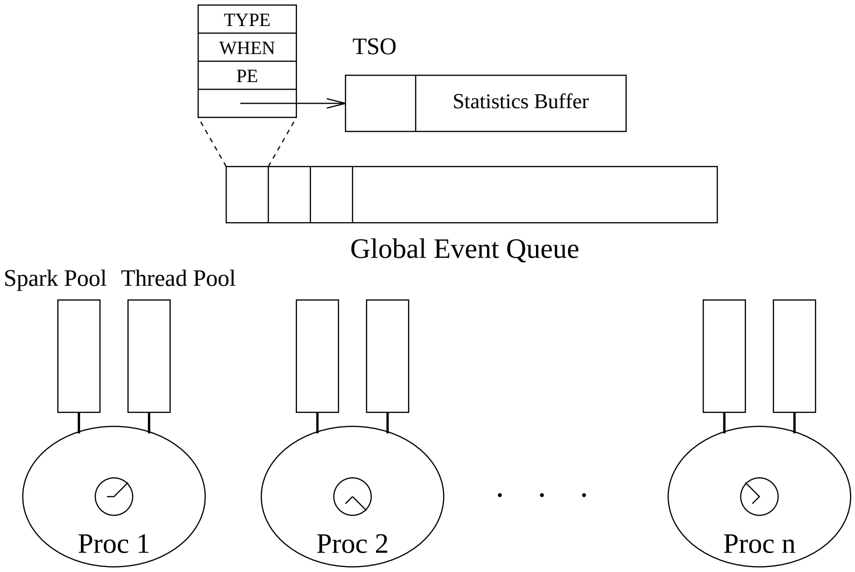


**Figure 3.1** Global structure of GranSim

events in the global event queue contain information about the type of the event, a time-stamp, the processor where it is happening and a link to the thread state object (TSO), a descriptor of the thread affected by the event. The statistics buffer accumulates important information such as the runtime, fetchtime, blocktime, amount of heap allocations etc.

From the presentation of the principles of parallel graph reduction in Section 2.3.1 it should be clear that the management of the spark and thread pools is fundamental for the behaviour of a parallel graph reducer such as GranSim. Therefore, we concentrate on the discussion of these two issues.

**Spark Management:** The spark pool holds sparks generated by threads on this processor as well as those obtained from other processors. By default it is managed as a first-in first-out (FIFO) queue. This means that older sparks appear earlier in the spark queue. Although this mechanism is likely to pick larger pieces of work first if the program has a divide-and-conquer structure, this is not necessarily the best way to manage the spark pool. Alternatives will be discussed in Section 5.5. In contrast to recent work on lazy threads (Goldstein et al. 1996), which tries to eliminate a separate spark pool altogether (see Section 5.7.1), such an explicit spark pool gives the runtime-system a handle to control the behaviour of the parallel program on a rather low-level, e.g. by attaching granularity information to individual sparks.

From the user's point of view two aspects of sparks deserve special attention. First of all, GranSim uses an evaluate-and-die model of computation, as discussed in Section 2.4.1. This means that one parallel thread may perform a reduction, for which another spark has been created. In short, sparks may be *subsumed* (Peyton Jones et al. 1987). This mechanism improves the granularity of the program to some degree. This issue is studied in greater depth in Chapter 5. Another important aspect of sparks is the fact that they may be *discarded* by the runtime-system. This is done for example when the closure, which should be evaluated, is already in weak head normal form (WHNF). It might also happen during garbage collection. For the programmer this means that he cannot rely on all sparks actually being turned into threads. This might be a problem if a spark is discarded although it drives the parallelism by generating many more sparks.

**Thread Management:** Each processor maintains a pool of runnable threads. Like the spark pool, the thread pool is implemented as a FIFO queue. The default scheduling algorithm for the threads is unfair: the currently running thread will only be descheduled if it demands a closure that is under evaluation by another thread or if it has to fetch remote data and asynchronous communication is enabled. If synchronous communication is turned on, the whole processor will be blocked while the data is fetched. In a previous version of GranSim a fair round robin scheduling mechanism was implemented. However, comparing simulations with these two variants of the scheduling mechanism showed only minor differences in the overall behaviour whilst increasing the simulation time significantly. The same unfair scheduling algorithm is also used in GUM.

A potential problem with unfair scheduling is that a single thread may exhaust all system resources. However, so far only the largest of our example programs, Lolita, causes such resource problems. Even in this case simulation time is a more serious limiting factor than resource exhaustion.

## 3.3 Characteristics of GranSim

This section discusses the main characteristics of GranSim, showing that the level of detail presented by the simulation supports a system-oriented view of parallel computation. In particular, the flexibility and the accuracy of the simulation will be discussed. Furthermore, a set of visualisation tools that have been implemented while developing GranSim proved to be crucial for a detailed analysis of the dynamic behaviour of the parallel programs.

The main characteristics of GranSim are

1. Support for different *levels of abstraction*;
2. *Flexibility* in simulating different parallel machines and different features of the runtime-system;
3. *Accuracy* of the simulation;
4. *Visualisation* of the dynamic behaviour and of the granularity of the program;
5. *Efficiency* of the simulation;
6. *Integration* of GranSim into a state-of-the-art optimising compiler (GHC);
7. *Robustness* of GranSim;
8. *Using Granularity Information* in the runtime-system.

Different levels of abstraction are provided by supporting both a GranSim-Light and a standard GranSim setup. In the latter configuration it is possible to abstract from certain aspects of the parallel execution, such as the communication latency, by setting the corresponding parameter to zero. This will become clear when discussing the simulation parameters in the following section. The GranSim User's Guide (Loidl 1996) contains a complete presentation of these parameters. A detailed discussion of the granularity improvement mechanisms in particular is given in Chapter 5.

### 3.3.1 Flexibility

GranSim enables the programmer to model a wide range of parallel architectures. This is possible by tuning many of the low-level characteristics of the parallel machine. For example the communication behaviour of a machine can be modelled by specifying several parameters like communication costs such as latency, message pack time etc, and the strategy that is used for packing a graph, such as incremental packing or bulk packing. The overhead imposed by the simulated runtime-system can be specified by setting costs for thread creation, context switch, etc. The specifics of the underlying processor can be changed, too (see Section 3.3.2).

Crucial for the flexibility of the simulator is its ability to simulate several different variants of important operations of the runtime-system. Variants of the most important operations in GranSim are:

- *Bulk fetching versus incremental fetching*: different *packing schemes* specify how much of a graph to pack into one packet.
- *Synchronous versus asynchronous communication*: different *rescheduling schemes* specify what to do while waiting for remote data.
- *Migration*: is a toggle indicating whether a runnable, but not running, thread may be moved ("migrated") to another processor. Experiments on GRIP have shown that migration, although very expensive, is essential for the performance of some programs (Hammond & Peyton Jones 1990). Migration is not implemented in GUM.
- Some of the more experimental features implemented in GranSim are: throttling communication by bounding the number of outstanding fetch requests, prefer stealing of threads over sparks, and prefer sparks of local closures over remote closures, to improve data locality.

The simulator is based on experiences from real parallel systems (GRIP, GUM) and therefore accurate in modelling aspects of the runtime-system. In fact, to a large extent GranSim shares the same code with GUM.

This close relationship between GranSim and GUM encourages the prototype implementation of runtime-system features not yet available in GUM. The author has used this possibility in implementing and measuring various packing schemes and various rescheduling schemes in GranSim (Loidl & Hammond 1996*b*), which are discussed in the following sections.

### Packing Schemes

A packing scheme prescribes how much of the graph to transfer to a processor that sends a fetch request for one closure. For example, an *incremental fetching* scheme only sends the closure that is immediately requested. This scheme aims to minimise the total number of closures that are sent during the execution of the program. This is achieved by fetching closures lazily when they are known to be required. However, this means that the requesting thread has to block for every remote closure, involving some delay determined by the latency of the machine. Such an incremental scheme has been used in the low-latency GRIP system.

In contrast, a *bulk fetching* scheme transfers a group of related closures in a single packet. The per-packet overhead is higher because packet construction and deconstruction are much more complicated. The gain is in reduced perceived latency per closure, because many nodes will be transferred in a single packet, and so will not need to be transferred individually if they are needed. As a refinement of this mechanism GranSim offers the possibility to specify a bound on the packet size or on the number of thunks that can be packed into a single packet. If neither limit is specified, all the graph that is reachable from the requested node will be packed into the packet. Note that packing multiple thunks into one packet essentially amounts to eager work distribution. The GUM implementation currently uses a full-subgraph packing scheme but imposes a limit on the packet size.

Figure 3.2 depicts the bulk fetching mechanism in action on a simple graph that involves sharing. The left hand side shows the graph before packing takes place, the right hand side shows the graph as it has been updated following packing. The centre of the diagram shows the packet that is constructed to transmit the graph. Shading is used to depict thunks, normal form closures are left unshaded. The packing algorithm traverses the graph structure in a breadth-first fashion. Each closure is given a global address which is used to preserve sharing both across the system and within the packet. When packing a thunk the original closure is overwritten with a *FetchMe* closure (lightly shaded), which acts as a global indirection to remote data

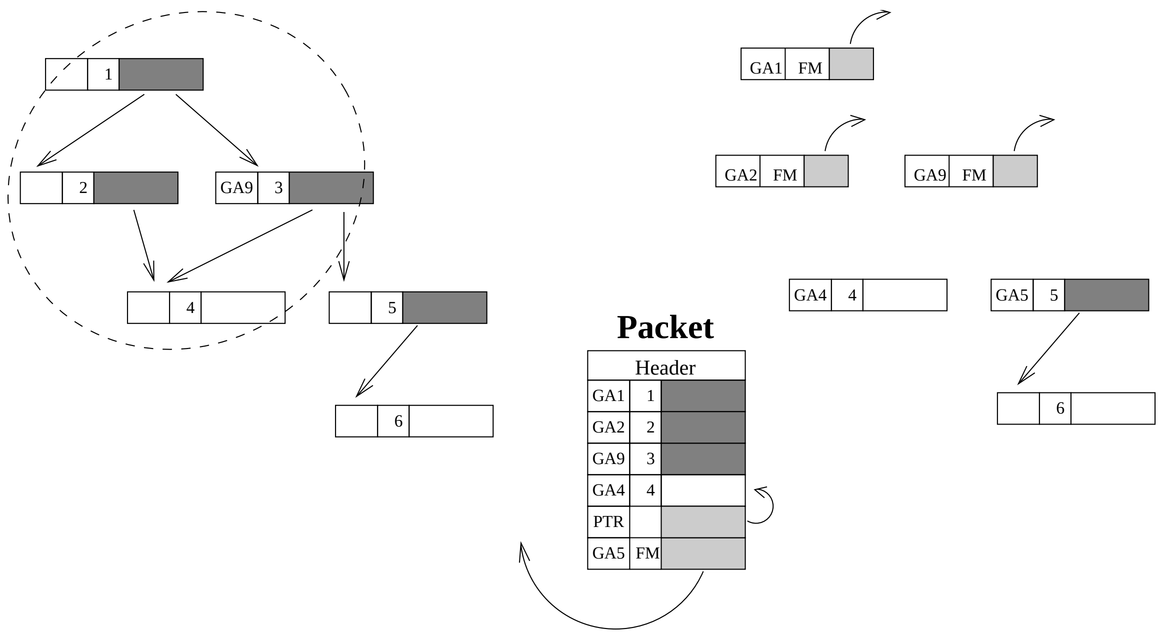


**Figure 3.2** The bulk fetching mechanism (with 3 thunks per packet)

(see Section 2.4.2). In contrast, normal form closures are duplicated by copying them into the communication packet.

The example in Figure 3.2 shows a packing scheme that packs a maximum of 3 thunks into a packet. Therefore one thunk is left behind on the original processor and is referenced by a *FetchMe* closure in the packet. A particularly useful version of this scheme is a normal-form-only packing scheme, which does not pack a thunk except for the root of the graph but it includes all normal forms before the first thunk because they can copied without duplicating work. The GUM system currently packs a full subgraph until one communication packet is filled.

### Rescheduling Schemes

A rescheduling scheme prescribes what the processor should do after having sent a fetch request to another processor. Two basic rescheduling schemes realise synchronous communication, where the processor waits for the remote data, and asynchronous communication, where another piece of computation is done in the interim. The latter amounts to latency hiding, since useful work can be performed until the requested data arrives.

Four different levels of rescheduling schemes specify how aggressive a processor will be in trying to obtain work:

1. only execute another runnable thread;
2. turn a spark into a thread if no runnable threads are available;
3. try to acquire a remote spark if the processor has no local sparks;
4. try to migrate another runnable thread if no remote sparks can be found.

These schemes are cumulative, so that thread migration will only be attempted if the three previous schemes have failed, etc. Note that the third and fourth 'global' rescheduling schemes will involve communication in order to obtain new work. In particular, the fourth scheme may introduce gratuitous thread migration towards the end of the computation, when the system load is low. The GRIP system uses synchronous communication and the GUM system currently tries to obtain remote sparks if no local work is available, corresponding to the third scheme in the list above.

### An Evaluation of Packing and Rescheduling Schemes

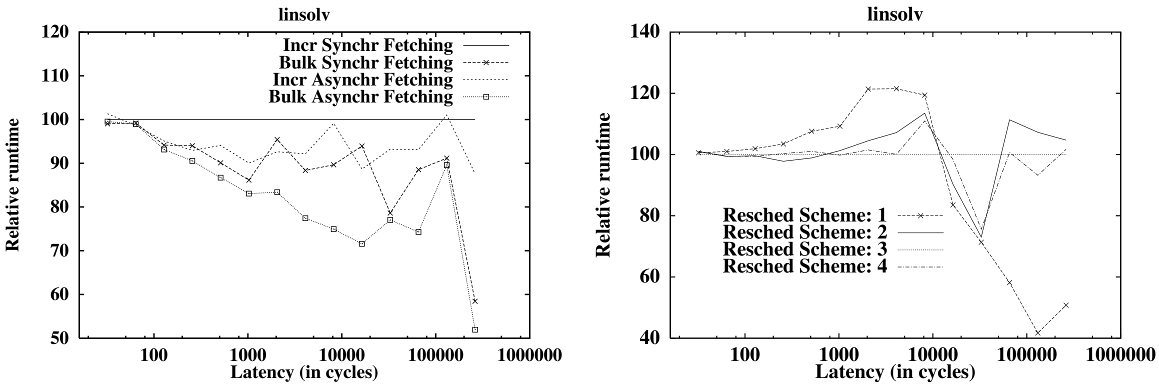


**Figure 3.3** A comparison of packing and rescheduling schemes

Figure 3.3 shows two of the measurements presented in Loidl & Hammond (1996*b*). The test program is the LinSolv algorithm discussed in Section 4.6. The left hand graph compares different packing schemes in combination with synchronous and asynchronous communication. The graph shows relative runtimes (in percent) with an incremental synchronous fetching scheme as the baseline. The best results are achieved when using a bulk fetching scheme with asynchronous communication. We observe a reduction of the total runtime of 17% and 28% for latencies between 1,000 and 50,000 cycles and a reduction of 50% at a latency of 260,000 cycles. The relative improvement in runtime increases for higher latencies. The graph also shows that bulk fetching should not be combined with synchronous communication because this would prevent the processor from performing useful work while waiting for the data.

The right hand graph of Figure 3.3 compares different rescheduling schemes with varying latencies. The baseline in this case is Scheme 3, which is used in GUM. This graph demonstrates that the best choice of a rescheduling scheme depends on the latency of the machine. For low latencies the more aggressive global schemes perform best since there is little cost associated with fetching work from remote processors. The improved load distribution outweighs the increased communication caused by a deteriorated data locality. However, for high latencies the dominant cost becomes that of moving data between processors. In this case, data locality is more important than an even load distribution. Therefore, the local rescheduling schemes usually perform better than the more aggressive schemes.

More detailed measurements with all different variants are presented and assessed in Loidl & Hammond (1996*b*). Several medium-scale programs have been used to test different packing and rescheduling schemes in setups with varying latencies. From these measurements the following conclusions can be drawn:

- *Rescheduling schemes:* For low latencies, where an even load distribution is more important than high data locality, aggressive rescheduling schemes deliver good work distribution and therefore good performance. For high latencies, however, the improved load distribution does not compensate for reduced data locality. The crossover point usually lies between 15,000 and 30,000 cycles, i.e. loosely-coupled multiprocessors.

- *Packing schemes:* In general, full-subgraph packing proves to be the best packing scheme. In practice, there is little danger that such a packing scheme will cause a disastrously uneven load distribution.

- *Thunk stealing:* Occasionally the full-subgraph packing scheme causes *thunk stealing*: the gratuitous offloading of thunks that will be needed later. This increases communication costs and hence reduces performance. We believe that thunk stealing is the reason for full-subgraph packing sometimes being worse than those schemes that pack a limited number of thunks per packet. This does not happen very frequently, however.

- *Bulk versus incremental fetching:* For low latencies (up to about 100 cycles) there is no difference in the performance of bulk and incremental fetching. Especially for very high latencies (more than about 50,000 cycles) bulk fetching achieves significant runtime improvements compared to incremental fetching even when using asynchronous communication for latency hiding.

- *Bounded packet size:* The average packet size is in general very small, even for full-subgraph packing (usually smaller than 15 closures). Therefore, changing the packet size, as has been previously suggested for improving communication performance, has hardly any effect on the runtime of the program.

As a result of the measurements in Loidl & Hammond (1996*b*) the following concrete suggestions for improving the GUM runtime-system can be made:

- For programs with a high degree of communication a normal-form-only packing scheme should be used in order to minimise a gratuitous transfer of work together with data ("thunk stealing"), which has been observed in GranSim measurements. It is probably not worthwhile implementing a more general scheme that allows the user to specify the number of thunks per packet because good values for such a parameter are very hard to predict.

- When running on a high-latency system of more than about 15,000 cycles a less aggressive rescheduling scheme should be used in order to maintain good data locality.

- In contrast to previous suggestions (Trinder, Hammond, Mattson Jr., Partridge & Peyton Jones 1996), we found that choosing a small packet size is not an effective means of tuning the granularity of the communication. This is due to the small average number of closures per packet in most programs.

### 3.3.2 Accuracy

To evaluate the accuracy of the simulation it is necessary to examine the accuracy of several key steps in the compilation and execution of a program. GranSim manages to achieve a balance in the accuracy of the following key steps:

- the compilation of the program;
- the simulation of the computation;
- the simulation of the communication.

**Compilation:** A prerequisite for achieving a high accuracy of the simulation is a compilation of the functional program, which avoids inefficiencies of a naive implementation of graph reduction. A naive compilation would distort every simulation because the compiled code, which is the input to the simulator, would differ significantly to code produced by an optimising compiler. Therefore the results even of an idealised simulation would have only a very limited relevance. GranSim is built on top of, and therefore makes use of, a state-of-the-art optimising compiler for Haskell (GHC). As a result the generated code is almost identical to the code used for sequential execution. The only difference is an instrumentation of the generated code on basic block level.

**Computation:** In order to assign computation costs to the basic blocks in the program an instruction count function is applied in an intermediate representation of the optimised program. This intermediate code bears a strong resemblance to low-level C without loops. At this level the operations in the program closely correspond to machine operations, which permits an exact modelling of the cost of computation. The instruction count function has been carefully tuned by analysing the assembler code generated by GHC and the results have been compared with the number of instructions executed in real Haskell programs. These comparisons have shown that the instruction count of the simulation lies within 10% for arithmetic operations, within 2% for load, store operations, within 20% for branch instructions and within 14% for floating point instructions of the real values (Hammond et al. 1995). Overall, it has to be emphasised that GranSim does not measure the computation in reduction steps, as it is often done in idealised simulators, but in machine cycles for a specific processor.

To permit different kinds of architectures to be modelled the instructions have been split into five classes, with different weights. The default weights in the following list model a SPARC processor and have been verified with Haskell programs in the sequential NoFib suite (Partain 1992), which is used to tune the Glasgow Haskell Compiler and which is publicly available (NoFib 1998). These weights are tunable in order to simulate other kinds of processors:

- arithmetic operations (default: 1 cycle),
- floating point operations (default: 1 cycle),
- load operations (default: 4 cycles),
- store operations (default: 4 cycles) and
- branch instructions (default: 2 cycles).

**Communication:** The basic communication parameters of a parallel machine such as latency, message creation costs, etc are parameters to the runtime-system. In total, GranSim offers 6 different parameters to describe the communication behaviour of a machine thus giving the user a high degree of flexibility in describing the characteristics of the machine being modelled. The accuracy of the modelled communication depends on the accuracy of the parameters provided by the user. One aspect of the communication that is not covered by GranSim is the topology of the parallel machine: in GranSim the latency between any two processors is the same. The latency also does not change with increasing network traffic. These shortcomings will be discussed in more detail in Section 3.5.

### 3.3.3 Visualisation

Together with the GranSim simulator a set of visualisation tools has been developed. Two kinds of profiles are generated: activity profiles and granularity profiles. This section discusses both kinds of profiles. These visualisation tools have proven indispensable in the parallelisation and optimisation of programs such as a linear system solver. Based on the group's experience from implementing several large programs (see Chapter 4), such tools are essential when working with a lazy language, in which the order of evaluation is not at all obvious from the program source.

All visualisation tools take a GranSim or a GUM profile, a log-file of the program execution, as input and generate a PostScript file as output. The format of this log-file is discussed in the GranSim User's Guide (Loidl 1996). Producing individual graphs can be seen as a form of static visualisation. Other packages such as the VISTA package (Halstead Jr. 1995) allow the user to step through the parallel execution based on the information available in the provided log file. This dynamic visualisation obviously can expose more information about the exact behaviour of the program. However, our experiences show that already static activity profiles with different levels of detail provide valuable information in order to tune the performance even of large parallel programs.

A promising direction of ongoing work is the use of cost centres, as developed for sequential profiling of Haskell (Sansom & Peyton Jones 1995), to connect points in the activity profiles with expressions in the source code. A prototype of combining GranSim with cost centre profiling, GranCC, to whose development the author has contributed, is already available (Hammond et al. 1997). Several projects for improving parallel profiling are aiming at increasing the information contained in these profiles, developing a self-describing log-file format that can be used for both sequential and parallel profiling, and developing graphical user-interfaces that provide a dynamic visualisation of the program behaviour. Research groups at the Universities of Glasgow, St. Andrews, York, the Open University and the Parallel Application Centre of the University of Oxford are collaborating in this effort.

#### Activity Profiles

The aim of the activity profiles is to summarise the activity of the machine during the computation in one graph. In order to give the programmer the possibility of examining the program execution in more detail, three different levels of detail are supported. Furthermore, it is possible to focus only on parts of the execution, like examining only one processor, by first applying a filter on the generated GranSim profile.

The activity profiles show the activity of the machine in three levels of detail:

- *Overall activity* of the whole machine;
- *Per-processor activity* of the individual processors;
- *Per-thread activity* of the individual threads.

The following subsections discuss each of these profiles and give examples.

**Overall activity:** The idea of the overall activity profile is to present a global picture of the computation. In particular, it should show the utilisation of the machine at each point. A drop in utilisation might reflect a performance bottleneck in the algorithm. This profile can be regarded as an "algorithm focusing" profile and is particularly important for an algorithm-oriented view of parallelism. The overall activity profile separates the threads into five different classes:

- running threads, i.e. threads that are currently performing a reduction, which are shown as a green area in the graph,
- runnable threads, i.e. threads that could be executed but that have not found an idle processor, which are shown as an amber area in the graph,
- blocked threads, i.e. threads that wait for a result that is being computed by another thread, which are shown as a red area in the graph,
- fetching threads, i.e. threads that are currently fetching data from a remote processor, which are shown as a light blue area in the graph,
- migrating threads, i.e. threads that are currently being transferred from a busy processor to an idle processor, which are shown as a dark blue area in the graph.

The overall activity profile in Figure 3.4 shows the number of threads in each class for each point in time. The example program in this case is a word search program, described originally in the FLARE book (Runciman & Wakeling 1995). It has a bottleneck at about 110k cycles. In the given setup, asynchronous communication with incremental fetching and a latency of 400 cycles, this results in a drop down to only one running thread for some time. As thread migration is enabled we observe several runnable threads being transfered to another processor immediately before that point. Overall this program suffers from a lack of parallelism, which can be

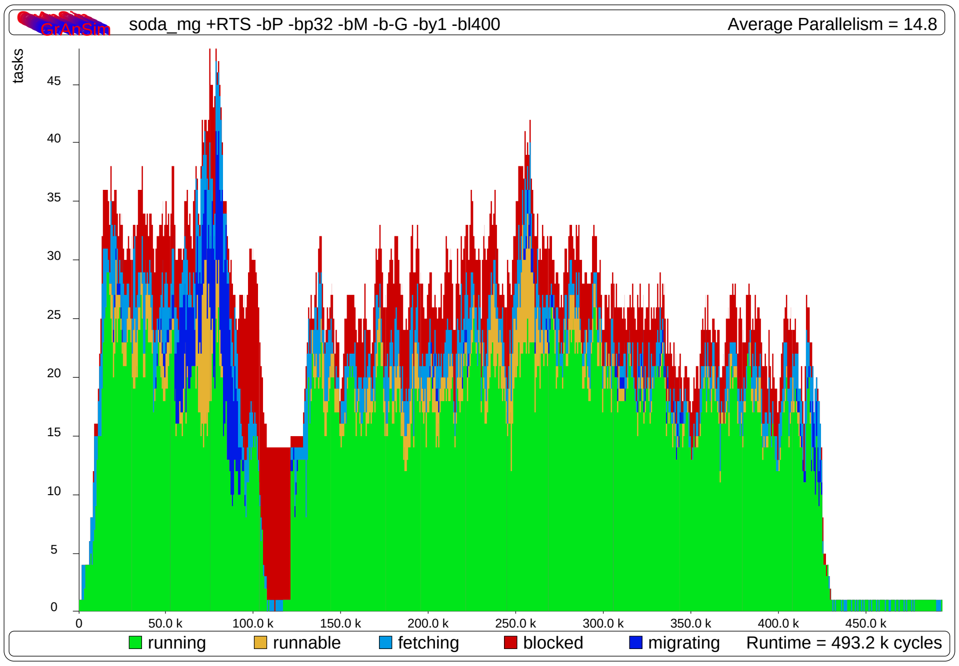


**Figure 3.4** Overall activity profile (original in colour)

seen from the low number of runnable threads although the machine rarely is fully utilised. The sequential tail of the program is due to the collection and the printing of the result. GranSim measures the costs of all Haskell input/output routines, which are written in a monadic style (Peyton Jones & Wadler 1993).

**Per-processor activity:** The idea of the per-processor activity profile is to show the most important pieces of information about each processor in one graph. Therefore it is easy to compare the behaviour of the different processors and to spot imbalances in the computation. This profile is often used to study runtime-system issues like the load balance in the system and is therefore most useful in a system-oriented view of parallelism. This profile can be regarded as a "load focusing" profile.

The per-processor activity profile shows one strip for each of the simulated processors. Each of these strips encodes three pieces of information:

- Is the processor *active* at a certain point? If it is active the strip appears in some shade of green (gray in the monochrome version). If it is idle it appears in red (white in the monochrome version).

- How high is the *load* of the processor? The load is measured by the number of runnable threads on this processor. A high load is shown by a dark shade of green (or grey).

- How many *blocked threads* are on the processor? This information is shown by the thickness of a blue (black) bar at the bottom of each strip. This bar may cover up to 80% of the strip. Thus, the load information is always visible "in the background".

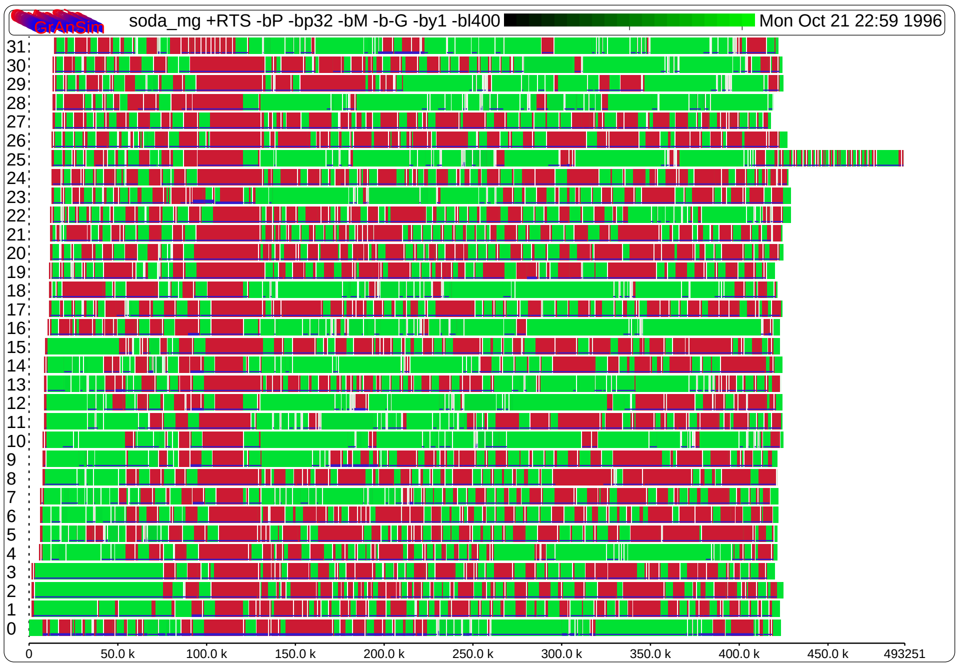


**Figure 3.5** Per-processor activity profile (original in colour)

The per-processor activity profile in Figure 3.5 uses the same example as in the previous section. The drop in utilisation at about 110k cycles is reflected by a rather large red area. The distribution of work at the beginning of the computation starts with low-numbered processors. Therefore, these processors have bigger pieces of work.

The distribution of work is quite even, which is represented as the same shade of green on all processors. The number of blocked threads is very small in general. Thread migration causes the main thread to be moved to processor 25, which is the processor that collects the final result.

Apart from showing the load of the processors, this kind of graph can also be used to show two additional pieces of information:

- *Migration:* This variant of the graph, a "migration" graph, shows arrows between processors indicating the migration of a thread from one processor to another. Load and blocking information are suppressed in this variant.
- *Sparking:* This variant of the graph, a "spark" graph, shows information about the number of sparks on a processor in the same way as the number of runnable threads, i.e. by shading. This graph is useful to highlight hotspots of spark creation.

**Per-thread activity:** The idea of the *per-thread activity profile* is to show the activity of all generated threads. For each thread a horizontal line is shown. The line starts when the thread is created and ends when it is terminated. The thickness of the line indicates the state of the thread. The possible states correspond to the groups shown in the overall activity profile. This profile can be regarded as a "thread focusing" profile.

The states of the threads are encoded in the following way:

- A *running* thread is shown as a thick green (gray) line.
- A *runnable* thread is shown as a medium red (black) line.
- A *fetching* or *migrating* thread is shown as a thin blue (black) line.
- A *blocked* thread is shown as a gap in the line.

This profile gives the most accurate kind of information. Although it is a static profile the information is so detailed that it is possible to "step through" the computation by relating events on different processors with each other. For example the typical pattern at the beginning of the computation is a running period for starting the thread

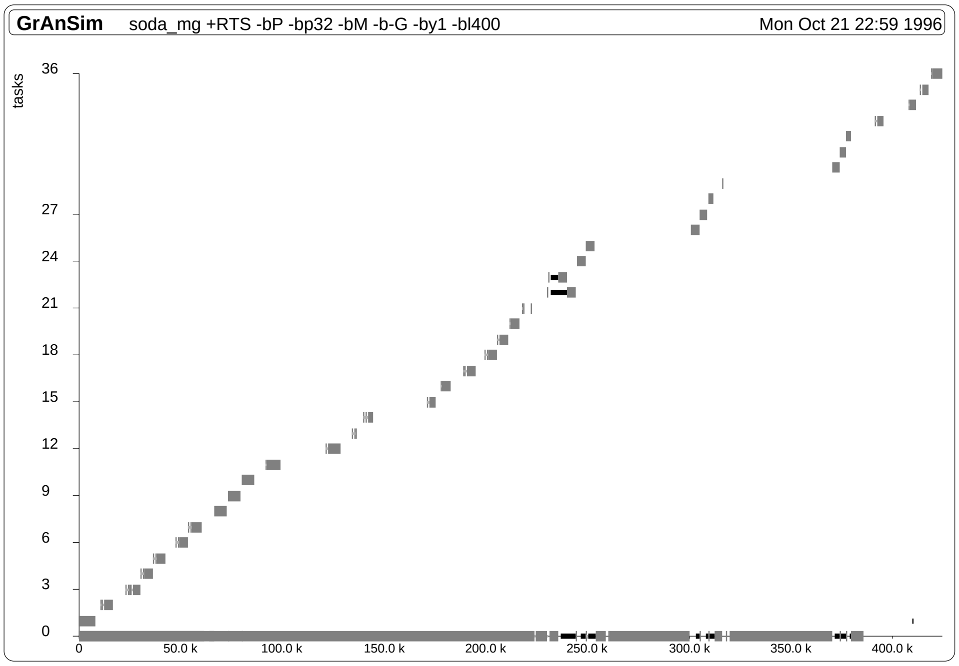


**Figure 3.6** Per-thread activity profile

followed by fetching remote data. After that the thread may become runnable, rather than running, if another thread has been started on that processor in the meantime.

The per-thread activity profile in Figure 3.6 only shows the threads that were executed on processor 0. As it is often done in practice, a filter has been used in order to obtain this kind of partial information. Usually this kind of profile is only used for focusing on a specific part of the execution or for a program with a rather small number of threads. The profile in Figure 3.6 shows the main thread, which is running most of the time. Occasionally it has to fetch data, shown as a thin line, or it is suspended because another thread is running on the processor, shown as a medium line.

### Granularity Profiles

The tools for generating *granularity profiles* aim at showing the total execution times of the generated threads. Of particular interest is the number of tiny threads, for which the overhead of thread creation is relatively high.

In order to show granularity information, i.e. information about the runtime of threads, two basic kinds of graphs can be generated:

- A *bucket statistics*, which collects threads with similar runtime in the same "bucket" and shows the number of threads in each bucket.
- A *cumulative statistics*, which shows how many threads have a runtime below a certain value. This graph gives more detailed information but is usually not necessary. Examples of using these graphs can be found in (Hammond et al. 1995).

**Bucket Statistics:** A bucket statistics partitions the x-axis, which represents thread execution times, into intervals and records the number of threads whose execution time lies in a specific interval. Thus, this statistics transforms continuous information, the runtime of a thread, into discrete information, the number of threads in a bucket. Standard methods for representing and processing of discrete data can be used on this data. For example, the number of threads in each interval is shown as a histogram. In order to show a wide range of possible values the y-axis is often shown in a log scale.

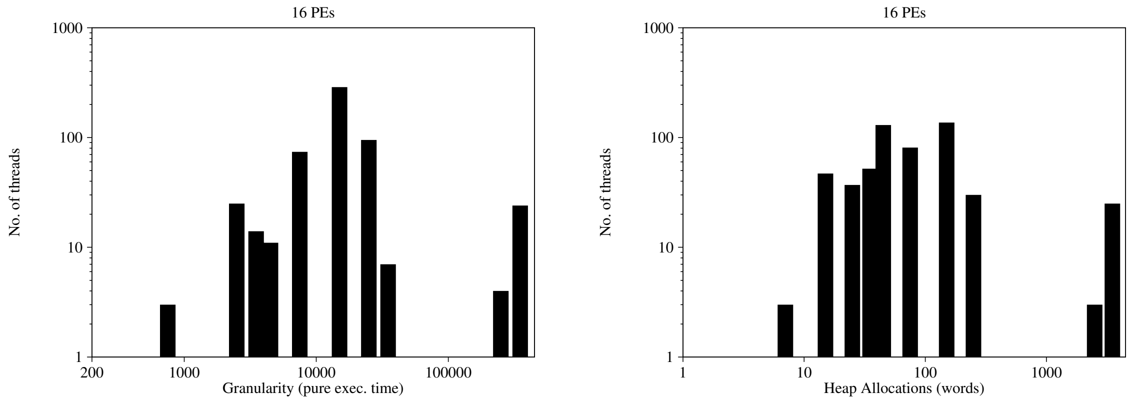


**Figure 3.7** Bucket statistics of thread runtime and heap allocations

Usually such a bucket statistics is used to analyse the distribution of the execution times of threads, giving a granularity profile. However, as can be seen in Figure 3.7

the same kind of statistics can be used in order to analyse different aspects of the execution such as the total amount of heap allocated by a thread. The similar profiles for both kinds of statistics in Figure 3.7 is typical for a range of programs we have studied. This reveals a non-obvious close relationship between the execution time and the number of closures allocated by a thread. Because the graph reduction model is centred around operations on the heap, it rarely happens that a time consuming thread performs very little allocation, even if the generated code has been optimised. As a matter of fact, our studies in (Hammond et al. 1995) show a more than 90% correlation between these two aspects for several example programs. The example program used in Figure 3.7 is again the word search algorithm.

**Cumulative Statistics:** One problem with the bucket statistics is that the resulting profile depends to some degree on the choice of the intervals. With an unlucky choice different results may show a similar profile. To avoid this problem, the visualisation tools can also generate cumulative statistics. In a cumulative granularity statistic a point $(x, y)$ in a graph indicates that $y$ threads have a runtime of at most $x$ cycles. Thus, the graph cumulates the number of threads and will show the total number of threads generated at the right end of the x-axis. This graph can be produced with either the absolute number of threads or the percentage of threads on the y-axis. Again the same kind of graph can be used to show aspects other than execution time.

### 3.3.4 Efficiency

The two most important features of GranSim for supporting a system-oriented view of the computation are its flexibility and accuracy (see Section 3.6 for a comparison with results from GUM). However, a high degree of accuracy also imposes a high bookkeeping overhead on the simulation. The three main factors governing the efficiency of the simulation are:

- the degree of communication in the program;
- the number of threads that are created; and
- the frequency of blocking a thread on a closure that is under evaluation.

The exact modelling of communication in GranSim is rather expensive, because each simulated communication causes a rather expensive context switch in the simulator. Such a context switch requires the current state of the simulator to be saved and restored. Furthermore, a "runtime-system call" has to be performed, interrupting the normal reduction process. This slows down the simulation especially of machines with low latency where much communication is performed.

Switching to another thread is also rather expensive. As a consequence, the total number of threads that are created affects the efficiency of the simulation in a crucial way. The influence of the number of threads on the performance of the simulation can be reduced by increasing the time slice given to each thread. This will result in a faster but less accurate simulation, because a thread may run ahead in the computation, ignoring communication events.

Another problem caused by a large number of threads is their heap consumption. With 30 words per thread, plus the size of the initial stack object, the heap used directly by the thread is not critical. However, because each thread holds on to a piece of graph, the total amount of live data can increase drastically. This causes more frequent garbage collections, which in turn increases the runtime of the simulation compared with an optimised sequential version. This point currently poses a problem for using the GranSim-Light setup in very large programs like Lolita.

In order to get an idea of the simulation costs Table 3.1 shows the simulation times, i.e. the time needed to run the simulation, of several programs run on GranSim and GranSim-Light with that on HBCPP (Runciman & Wakeling 1993), an idealised simulator for the same source language. As example programs a set of non-trivial programs from the emerging parallel NoFib suite has been used: a ray tracer, Ray, the same word search program, Soda, that has been used as an example for the visualisation tools, a linear system solver, LinSolv, discussed in detail in Section 4.6, a determinant computation, Determinant, used as a part of the linear system solver, and a matrix multiplication, MatMult. Two values are given for the GranSim simulation times: the first value uses the default time slice given to every thread; the value in parentheses uses a very small time slice for a more accurate but slower simulation. The difference from the first value gives an idea how much the simulation time can be tuned by choosing a different time-slice.

In three cases, Ray, LinSolv, and MatMult the GranSim-Light setup shows a significant higher runtime compared to the standard GranSim setup. This is mainly due

**Table 3.1** Simulation times (in seconds) of GranSim and HBCPP

| Program | GranSim default time slice | GranSim short time slice | GranSim-Light | HBCPP | optimised GHC |
|---|---|---|---|---|---|
| Ray | 70.7 | 198.9 | 141.3 | 73.2 | 11.1 |
| Soda | 2.4 | 5.5 | 1.5 | 0.8 | 0.1 |
| LinSolv | 75.9 | 96.8 | 334.0 | — | 0.1 |
| Determinant | 7.9 | 8.4 | 4.3 | 4.1 | 1.7 |
| MatMult | 22.3 | 26.9 | 65.9 | 26.9 | 0.4 |

to the large number of threads that are created in the idealised simulation, causing a large number of context switches. This aspect is elaborated further in Section 3.4. It should be noted that faster simulation time is not the main goal of GranSim-Light. Often it generates a faster simulation but the main purpose is to simulate an idealised machine, reflecting an algorithm-oriented view of parallelism.

Usually, GranSim is between 1.5 and 2.5 slower than HBCPP, the factor would probably be larger for `linsolv` but this program did not compile successfully under HBCPP. Considering the additional information produced in the standard GranSim setup this can be regarded as an acceptable factor. In the case of MatMult and for some very small example programs it occasionally even manages to outperform HBCPP. One reason for the reduced simulation time might be the improved code generation. Because GranSim is integrated in GHC we can profit from the ongoing tuning of the compiler itself (see the following section for details).

Compared to an optimised sequential version the simulation shows a slow-down of a factor of 4.6 to 759. Again the worst case is generated by `linsolv` with an abundance of parallel threads and a lot of communication in the program. Most of the simulations exhibit a slow-down of 10 to 15. Considering that GHC produces the fastest code of all available Haskell compilers (Hartel 1995), these factors still render the simulator useful for large programs and this has been proven for programs such as LinSolv (Section 4.6) and Lolita (Section 4.5).

### 3.3.5 Integration into GHC

GranSim is built on top of the Glasgow Haskell Compiler (GHC), a state-of-the-art optimising compiler for Haskell. This means that the execution of sequential code in the simulator is realistic. In fact, the code generated by GranSim is almost identical to the sequential code generated by GHC. The only difference are macros that check for the existence of a closure on a processor, at the beginning of every basic block, and another macro for adding the execution time of the basic block to the local clock, at the end of this basic block.

It is possible to use all the features of a normal GHC compilation in GranSim, too. For example, the `ccall` mechanism can be used to call C functions in a parallel program. This feature is essential for the parallelisation of Lolita(see Section 4.5). With this mechanism optimised sequential, possibly even imperative, code in libraries can be called from a parallel lazy functional program. This feature has been exploited in an experimental implementation of a parallel resultant algorithm using basic polynomial operations of a sequential computer algebra library.

One of the main features of GHC is the use of many program transformations in order to optimise the sequential code. This covers well-established optimisations such as inlining and the use of strictness information as well as rather new optimisations such as let-floating and deforestation. The influence of these new sequential optimisations on the parallel execution of a program is an interesting but largely unstudied area. For example deforestation might eliminate intermediate lists that are crucial for the parallel execution of the program. Indeed Santos reports that in one example program (Fast Fourier Transformation) the full laziness transformation creates a sequential bottleneck, which slows down the computation by a factor of 6 to 10 (Santos 1995, Section 5.2.2). GranSim would seem to be the ideal basis for studying these interactions in more detail.

### 3.3.6 Robustness

The robustness of GranSim has been proven by using it in the parallelisation and performance tuning of a set of large Haskell programs. Some of these programs are discussed in more detail in Chapter 4. Most of the parallelisation of the Lolita natural language engineering system has been done by using GranSim. Other scientists have used GRANSIM to parallelise substantial pieces of Haskell code such as a program that determines accident blackspots based on a large database of traffic accident reports (Wu & Harbird 1996, Trinder et al. 1998) and Naira, a parallelising compiler for a subset of Haskell (Junaidu 1998).

Without a simulator it would be much more difficult to parallelise such large programs because of system issues, e.g. integrating foreign language calls, and "external" aspects of the execution, e.g. system load, cannot be easily eliminated. The separation into GRANSIM and GRANSIM-Light configurations encourages the parallel program to be developed in two stages: first the parallel algorithm is developed in a machine independent setting; then it is optimised for a specific machine. In particular, the parallelisation of Lolita showed the importance of having a simulator that is integrated in a state-of-the-art-compiler with all its tools: it was crucial to have a profiler for the sequential version of the program. Based on these experiences of using both GRANSIM and GUM in the parallelisation of several programs the parallel programming group at Glasgow has developed a parallelisation methodology, with GRANSIM as one of its major components (see Section 4.8).

## 3.4 GRANSIM-Light

One main purpose of GRANSIM is to provide a testbed for variations of the runtime-system. This requires a very accurate simulation that is flexible enough to model different kinds of parallel architectures. However, in early stages of the development of a parallel algorithm a more abstract view of parallel computation is advantageous. This different attitude requires slightly different characteristics of the simulator.

The GRANSIM-Light setup has been designed to satisfy such an *algorithm-oriented* view of parallelism. Therefore, GRANSIM-Light models an idealised machine with

- an infinite number of processors and
- zero communication costs.

This difference in modelling the parallel execution of a program requires changes in the structure of the simulator. Most importantly, the spark and thread pools are not

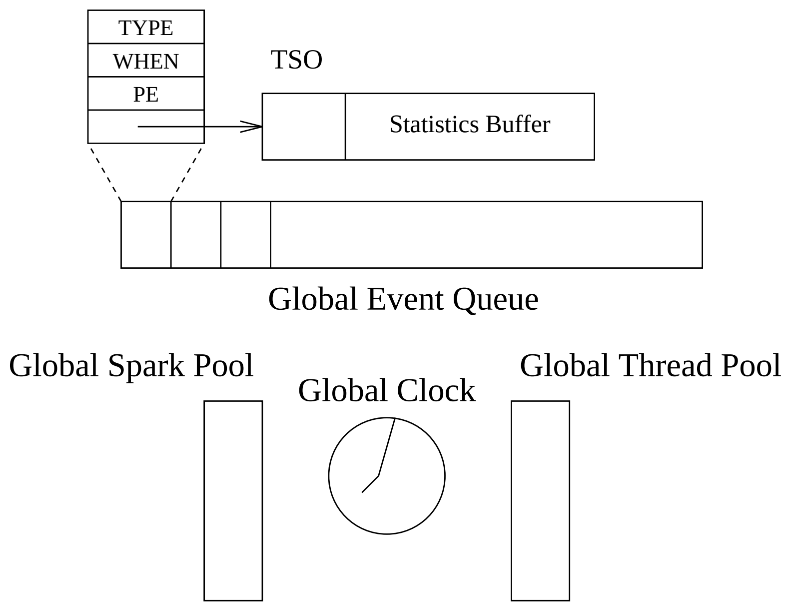


**Figure 3.8** Global structure of GranSim-Light

distributed in this setup. Figure 3.8 shows the global structure of the GranSim-Light setup.

This setup exposes all parallelism in the algorithm and allows the programmer to tune the performance of the algorithm before studying its dynamic behaviour on a specific parallel machine. Although such a simulation gives a less accurate picture of the parallel behaviour on a concrete machine, it has proven to be an important step in the methodology for parallelising large lazy functional programs (see Section 4.8).

The GranSim-Light setup is very close to the HBCPP simulator (Runciman & Wakeling 1993). In Section 3.6 we compare the results of some simulations under both simulators. Table 3.1 has already shown that the simulation time in GranSim is comparable to that in HBCPP. GranSim-Light sometimes manages to be as fast as HBCPP and is within a factor of 2.5 for the remaining programs.

One problem with GranSim-Light, however, is the fact that its performance depends very much on the number of generated threads. The idealised simulation of GranSim-Light usually creates a much larger number of threads than the standard simulation because in the latter case the evaluate-and-die mechanism manages to subsume potential parallel threads. Clearly, the evaluate-and-die mechanism cannot be effective in a setup where every spark is immediately turned into a thread. We have seen this behaviour when comparing simulation times in Table 3.1. For some example programs in these measurements the GranSim-Light simulation is significantly slower than the standard setup of GranSim. The main reason for this slow-down is the large number of context switches necessary to simulate the graph reduction of and the interaction between so many threads.

## 3.5 Shortcomings of GranSim

Despite the high degree of parameterisation of GranSim, there are certain aspects of a parallel machine that are not modelled. This section comments on these shortcomings and their impact on the development of parallel algorithms.

**Computation:** GranSim models an execution on a homogeneous MIMD multiprocessor. This model does not include the concept of clusters of processors, with cheap local communication. Nor does this model encompass SIMD machines, which operate with only a single instruction stream. However, this model corresponds to the underlying computation model of GUM.

**Communication:** Two of the most important aspects of a parallel machine that are not covered by GranSim relate to the communication behaviour of the machine: the bandwidth of the communication and the topology of the underlying machine. GranSim assumes that the latency between two processors is independent of the communication traffic. In reality, however, "contention" will occur at some point, drastically degrading the performance of the communication. However, this usually only happens with an excessive amount of communication and should therefore be no problem for normal executions. Another simplifying assumption is that the distance between any two processors in the system is the same. This fixes the simulation to one special topology, a fully-connected graph. However, experiences with modern parallel machines show that the topology has a rather small influence on the communication speed.

**Memory Management:** One important aspect in the execution of a parallel program is the data locality. In the computation model used by GranSim as well as GUM there is only very limited support for studying this aspect. An experimental feature in GranSim allows absolute placement of a process on a specific processor, but the data will always travel to the thread never vice versa. A useful extension of GranSim would be the implementation of "sticky closures" that have to be evaluated on the processor on which they have been created. The idea of such an implementation would be to automatically create a spark for a sticky closure when it is demanded. The usual runtime-system mechanisms can then be used to turn the closure into a thread and to evaluate it. This evaluation must be on the specified processor but the runtime-system still has the choice to discard the spark.

GranSim does not provide any modelling of garbage collection in the parallel system. The main motivation for this design decision is that the choice of one particular mechanism would likely have global effects in the execution, e.g. reference counting garbage collection introduces an overhead when copying any closure in the system. Thus, all results would be biased towards the chosen form of garbage collection.

**Extensions of GranSim:** One important shortcoming is the lack of a parallel profiling mechanism. When parallelising big programs it would be very important to mark certain threads that are of special interest and to focus on these threads with the visualisation tools. So far, only a rudimentary thread marking mechanism has been implemented. It propagates a thread name to all children and makes it possible to change the name during execution. In order to use this information special filter programs have to be applied to the GranSim profile. In the meantime, a parallel profiler, GranCC, has been constructed by merging GranSim with sequential cost centre profiling (Hammond et al. 1997). Initial results of this research effort, to which the author is contributing, show valuable additional information. An alternative approach is to dynamically mark evaluation strategies (see Section 4.3) in the code to provide information about which threads have been generated by which strategy. This approach is currently pursued by a research group at The Open University.

## 3.6 Validation of Simulation Results

This section gives a validation of some simulation results by comparing profiles obtained from GranSim with those from HBCPP, GRIP, and GUM. This comparison shows that GranSim yields a realistic picture of a program's parallel behaviour, provided that the GranSim parameters are set to model the underlying hardware architecture.

### 3.6.1 GranSim versus HBCPP

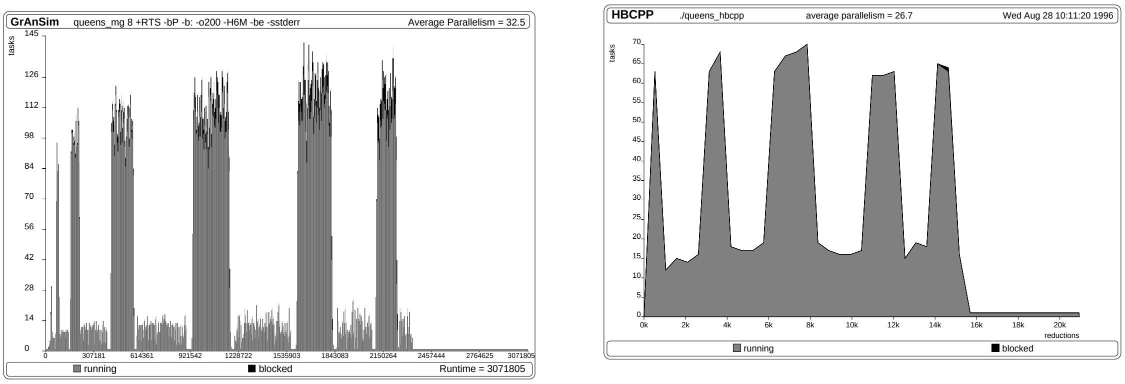


**Figure 3.9** Activity profiles from GranSim and HBCPP

Figure 3.9 compares the overall activity profiles for the `queens` program generated by GranSim and HBCPP. The activity profile produced by the GranSim execution is significantly more detailed, which results in a more fine-grained picture. It also manages to exhibit stages of blocking that are too short to be detected in HBCPP. Most importantly, the overall pattern of the computation is the same.

### 3.6.2 GranSim versus GRIP

Section 4.6 discusses three variants of a symbolic algorithm for solving a system of linear equations, LinSolv. Starting with a rather inefficient algorithm the performance of this algorithm is tuned and sequential bottlenecks are eliminated. The final algorithm has also been executed on the GRIP multi-processor. Figure 3.10 presents a comparison of the activity profiles generated by GranSim and GRIP. The GranSim version uses a setup with 16 processors and a latency of 400 cycles, matching the GRIP configuration. In the GRIP profile no runnable threads are shown because this kind of information is not collected. The shape of both profiles is very similar. Both profiles show a small peak of parallelism at the end of the computation. Comparing the raw numbers we observe an average parallelism of 15 under GranSim, whereas the average parallelism on GRIP is 14.5. The speedup obtained under GranSim, 11.92, is slightly below the speedup on GRIP, 13.58.

The most pronounced difference is the larger number of blocked threads in GranSim. This is probably due to the use of local sparking in GRIP, which is not simulated in GranSim. Local sparking distinguishes between local spark pools for each processor and one shared global spark pool. In order to improve data locality local sparks are preferred. Only in the case of a global shortage of sparks are the local sparks moved into the global spark pool. In this example the GranSim graph shows that there are runnable threads through most of the computation. Therefore, the GRIP version will rarely have to move sparks into the global spark pool, where they can be picked up by other idle processors. In total this leads to a smaller number of generated threads.

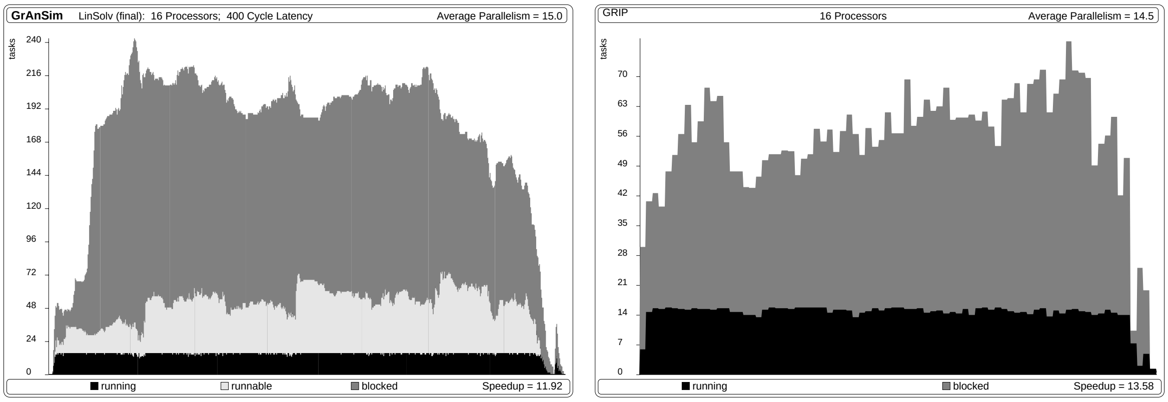


**Figure 3.10** GranSim and GRIP activity profiles of LinSolv

One of our example programs that shows a very interesting granularity profile is a parallel ray tracer. This program has been developed by Kelly in his thesis (Kelly 1989).

Hammond et al. (1994) study the granularity of this algorithm on the GRIP multiprocessor, deriving granularity profiles for each of the programs. The author used the same code of the ray tracer under GRANSIM to analyse the granularity of the generated threads. Figure 3.11 compares these profiles in a bucket statistics. In both cases a log scale is used to show even small buckets. GRANSIM measures time in machine cycles, whereas in the GRIP measurements the granularity is measured in terms of the number of heap allocations. The previously mentioned high correlation between execution time and heap allocations justifies this approximation of execution time. This program shows two main clusters of threads with respect to their runtimes: two clusters of short threads and a cluster of large threads. The short threads represent processes that "drive the parallelism" in the program, generating many sub-threads. The large threads are performing the actual computation. Because of the different measure of execution time, concrete x-values cannot be directly compared. However, the granularity profile in both cases is the same.

### 3.6.3 GRANSIM versus GUM

Figure 3.12 gives a comparison of a parallel determinant computation executed under GRANSIM, left hand side, and on a Sun SPARCserver shared memory machine with four processors under GUM, right hand side. The overall shape of both profiles exhibits a very similar overall behaviour of the program. The GRANSIM version underestimates the number of blocked threads and especially the number of fetching threads. The latter is a general trend, which can also be observed in the Lolita system (see Section 4.5). Although the overhead for creating a communication packet has been increased in this simulation it does not model all of the software overhead in PVM, which is in part data dependent. The regular, short drops in the utilisation of the GUM profile may in part be caused by operating system interference, because the 4 processor machine used in this experiment has a significant load of users.

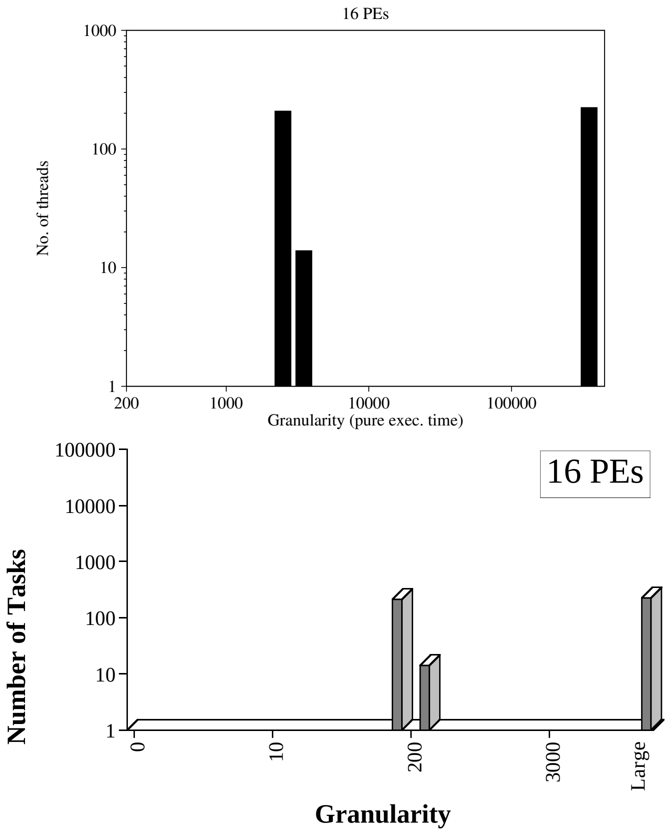


**Figure 3.11** GranSim (top) and GRIP (bottom) granularity profiles of a ray tracer

## 3.7 Summary

A simulator for the parallel execution of functional programs may be of use for either the programmer, who wants to study the parallel behaviour of a certain algorithm, or the compiler designer, who wants to study the effectiveness of certain mechanisms in the runtime-system. This chapter has shown that GranSim is a useful tool for both groups by supporting a high-level algorithm-oriented view as well as a low-level system-oriented view. In the latter view the focus might be on an extremely accurate simulation of a specific machine or on a flexible simulation of a wide range of parallel architectures. GranSim supports the approach of a flexible simulation by being highly parameterised without losing accuracy on the compilation level. Only certain very low-level machine characteristics are not captured in the simulation. Taking such a system-oriented view, GranSim measurements with implementations

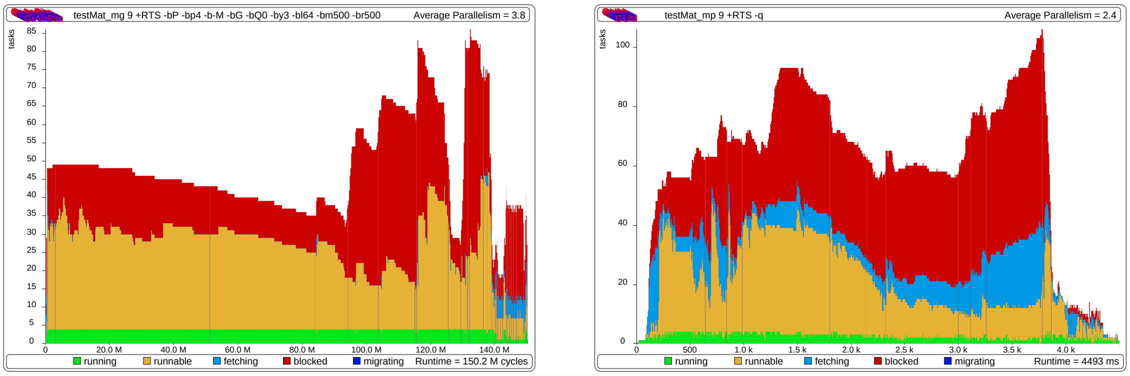


**Figure 3.12** GranSim and GUM activity profiles of a determinant computation

of alternative packing and rescheduling schemes have led to concrete suggestions for improving the GUM runtime-system for specific architectures, e.g. by packing smaller graph structures in highly-communicating programs or by using a less aggressive rescheduling scheme in high-latency systems.

In the following chapter GranSim will be used in the parallelisation and performance tuning of a set of large functional programs. This will demonstrate its practical usefulness beyond its original design as a testbed for implementing variants in the parallel runtime-system. The integration of GranSim into a parallel engineering environment together with the GUM parallel runtime-system, and the availability of visualisation tools in both systems are crucial in the development of large parallel programs.

# *Chapter 4*

# Large-Scale Parallel Functional Programming

**Capsule**

The superior computational power of parallel machines is most likely to be used in time consuming programs. Such programs are typically large. During the performance tuning of the parallel code it is often necessary to restructure parts of the code. For these reasons, a modular design is even more important for parallel programs than for sequential programs. Lazy functional languages offer a high level of modularity via higher-order functions and a non-strict semantics. This chapter focuses on the question how to specify parallelism in a lazy functional language without sacrificing modularity.

Previous experiences with writing medium-scale parallel programs have shown that the undisciplined use of `par` and `seq` annotations in the program can yield opaque code. This observation has led to the development of evaluation strategies based on laziness, overloading, polymorphism, and higher-order functions. This chapter presents evaluation strategies, which have been developed in a group effort, and contributes to the design of strategies by augmenting the core module with a construct for strategic function application. The resulting module has been used in parallelising several large programs including LinSolv, a linear system solver, an Alpha-Beta search algorithm, and Lolita, a natural language engineering system consisting of more than 47,000 lines of Haskell. These programs show that with only a few localised changes in the code good parallel performance can be achieved in programs that have not necessarily been written with parallel execution in mind. The laziness of the language favours a data-oriented style of parallel programming, where the parallelism is defined on intermediate data structures rather than within specific modules of the program. This facilitates top-level parallelisation and restricts the contextual knowledge the programmer has to have about the program.

## 4.1 Introduction

Although the advantages of the high level of abstraction in functional languages mainly show up in big programs there is a daunting shortage of such programs. In the context of parallel processing this is even more critical since realistic time-consuming programs, which should be executed in parallel, are often large. Obtaining a parallel version that exhibits a reasonable parallel performance without spending a lot of effort in modifying the code is therefore of utmost importance.

This thesis focuses on symbolic computation as main application area. By and large, programs in this area use the major advantages of functional languages such as higher-order functions and algebraic data-types much more heavily than numerical computation programs. Thus they are a natural application for functional languages. For programs with these characteristics it is possible to make use of parallel computation without a vast effort in recoding the program, even if that results in the loss of some parallelism. Again this is in contrast to the approach towards parallel computation usually taken for numerical applications, where it is feasible to invest a lot of time in parallelising one particular program. In contrast, the parallelisation of the programs in this chapter takes an approach of "acceptable gain for low pain".

In order to cope with large programs the parallel programming group at Glasgow has developed *evaluation strategies*, a new programming technique based on lazy evaluation, overloading, polymorphism, and higher-order functions. Evaluation strategies allow a clean separation of algorithmic code from an operational description of the parallel program behaviour. This chapter discusses the author's contribution to the development of strategies and his parallelisation and performance tuning of several large functional programs. This presentation shows that the parallel program development is much easier when using strategies, in particular because of better support for modularity, and that most of the complexity of parallel program development for imperative languages is absent in this model, because synchronisation and communication are managed entirely by the runtime-system.

The parallel programming group at Glasgow has studied about 8 medium to large parallel functional programs. This chapter describes the following three programs:

- LinSolv, a program for finding an exact solution of a system of linear equations. It is interesting for its use of an approach typical to many algorithms in computer algebra.
- Alpha-Beta search, a program for performing a heuristic search in a tree structure, usually used in game programming. It is a typical program for AI applications.
- Lolita, a large natural language engineering system. It is a very general system and can be used for extracting semantics from newspaper articles, translate text between languages, or for interactive language tutoring.

The contributions of this thesis to the work described here are as follows. The author's experience with parallelising LinSolv has initiated the development of evaluation strategies in a group effort led by Dr. Phil Trinder. The author's main independent contribution to the design of strategies is the development of strategic function application as a convenient way to express pipeline parallelism and to combine it with other forms of parallelism via function composition. The modified strategies module has been used in the author's parallelisation of LinSolv, strategy version, and of Alpha-Beta search, based on the sequential algorithm by Hughes (1989). These experiences have led to changes in the core design of evaluation strategies. The parallelisation of Lolita has been done in collaboration with the Natural Language Engineering Group at the University of Durham. Sections 4.2, 4.3, and 4.9.1, describing evaluation strategies, are revised versions of material published in Trinder et al. (1998). Sections 4.4 and 4.5 cover material published in Loidl & Trinder (1997) and Loidl et al. (1997), respectively. Section 4.6 is a revised version of material submitted for publication in Loidl (1997).

The structure of this chapter is as follows. Section 4.2 discusses problems when using annotations in order to describe parallel program behaviour for large programs. Section 4.3 introduces evaluation strategies and presents simple generic strategies demonstrating the flexibility of this approach. The following three sections present case studies of using strategies on several large programs: an Alpha-Beta search algorithm in Section 4.4, Lolita, a natural language engineering system in Section 4.5, and

LinSolv, a linear equation solver in Section 4.6. Section 4.7 compares the style of parallel programming in a lazy functional with a strict imperative language. Section 4.8 outlines a methodology for parallelising large lazy programs based on the acquired experiences with parallelising several large applications. Finally, Section 4.10 concludes.

## 4.2 Problems with Parallel Programming in-the-large

The big advantage of functional programming languages is the fact that they avoid overspecification by only defining the result without specifying an exact order of evaluation steps. More informally, they specify *what* to compute without fixing *how* to compute it. However, when writing an explicitly parallel program it is necessary to specify some aspects of the dynamic behaviour of the program. In the model used in this thesis this means exposing parallelism by marking expressions that might be evaluated in parallel. Since the basic execution model is a lazy one, the programmer may also want to specify the evaluation degree in the program in order to guarantee a certain amount of evaluation without relying on the quality of the strictness analyser.

This approach abstracts from details about thread creation, thread placement, synchronisation, data transfer, and many other aspects that often have to be explicitly handled in a parallel language by the programmer. However, even just describing potential parallelism together with evaluation degree may lead to a program that is cluttered with behavioural code. The undisciplined use of annotations in the parallelisation of several programs, such as a linear system solver, has generated opaque parallel code. The comparison of a straightforward parallelisation of LinSolv with a version using strategies in Section 4.6 shows the practical advantages of a more structured approach towards exposing parallelism.

### 4.2.1 A Simple Example

As a simple example demonstrating the problem mentioned above, let us consider parallel quicksort. A naive version of a parallel function might be written as follows.

```
quicksortN :: (Ord a) => [a] -> [a]
quicksortN []     = []
quicksortN [x]    = [x]
quicksortN (x:xs) = losort `par`
                    hisort `par`
                    losort ++ (x:hisort)
                    where
                      losort = quicksortN [y|y <- xs, y < x]
                      hisort = quicksortN [y|y <- xs, y >= x]
```

The intention is that two threads are created to sort the lower and higher halves of the list in parallel with combining the results. Unfortunately `quicksortN` has almost no parallelism because threads in GPH terminate when the sparked expression is in weak head normal form (WHNF). In consequence, all of the threads that are sparked to construct `losort` and `hisort` do very little useful work, terminating after creating the first `cons` cell. To make the threads perform useful work a "forcing" function, such as `forceList` below, can be used. The resulting program has the desired parallel behaviour, yielding a parallel divide-and-conquer structure. However, the definition of `quicksortF` is cluttered with behavioural code, namely the forcing functions.

```
forceList :: [a] -> ()
forceList [] = ()
forceList (x:xs) = x `seq` forceList xs

quicksortF []      = []
quicksortF [x]     = [x]
quicksortF (x:xs)  = (forceList losort) `par`
                     (forceList hisort) `par`
                     losort ++ (x:hisort)
                     where
                       losort = quicksortF [y|y <- xs, y < x]
                       hisort = quicksortF [y|y <- xs, y >= x]
```

### 4.2.2 Data-Oriented Parallelism

Quicksort is an example of (divide-and-conquer) *control-oriented* parallelism where subexpressions of a function are identified for parallel evaluation. *Data-oriented parallelism* is an alternative approach where elements of a data structure are evaluated in parallel. A parallel map is a useful example of data-oriented parallelism; for example the `parMap` function defined below applies its function argument to every element of a list in parallel.

```
parMap :: (a -> b) -> [a] -> [b]
parMap f [] = []
parMap f (x:xs) = fx `par` fxs `seq` (fx:fxs)
                  where
                    fx = f x
                    fxs = parMap f xs
```

The definition above works as follows: `fx` is sparked, before recursing down the list (`fxs`), only returning the first constructor of the result list after every element has been sparked. Note that if the function argument supplied to `parMap` constructs a data structure, it must be composed with a forcing function in order to ensure that the data structure is constructed in parallel.

### 4.2.3 Dynamic Behaviour

As the examples above show, a parallel function must describe not only the algorithm, but also some important aspects of how the parallel machine should organise the computation, i.e. the function's dynamic behaviour. In GpH, there are several aspects of dynamic behaviour:

- *Parallelism control*, which specifies what threads should be created, and in what order, using `par` and `seq`.
- *Evaluation degree*, which specifies how much evaluation each thread should perform. In the examples above, forcing functions were used to describe the evaluation degree.
- *Thread granularity*: it is important to spark only those expressions where the cost of evaluation greatly exceeds the thread creation overheads.
- *Locality*: part of the cost of evaluating a thread is the time required to communicate its result and the data it requires, and in consequence it may only be worth creating a thread if its data is local. Because GpH does not contain explicit placement information, locality has to be controlled indirectly, e.g. by constructing data structures that contain all data that should be kept local.

Evaluation degree is closely related to strictness and defined over the same partially ordered, lifted domain of values. If the evaluation degree of a value in a function is less than the program's strictness in that value, i.e. its value in the semantic domain is smaller than that defined by its strictness property, then the parallelism is conservative, i.e. no expression is reduced in the parallel program that is not reduced in its lazy counterpart. In several programs we have found it useful to evaluate some values speculatively, i.e. the evaluation-degree may usefully be more strict than the lazy function. Although this runs the risk of performing unnecessary computation it allows the programmer to specify parallelism that is useful most of the time.

## 4.3 Evaluation Strategies

### 4.3.1 Evaluation Strategies

In the examples above, the code describing the algorithm and dynamic behaviour are intertwined, and as a consequence both have become rather opaque. In larger programs, and with carefully-tuned parallelism, the problem is far worse. This section describes *evaluation strategies*, a solution to this dilemma. The driving philosophy behind evaluation strategies is that *it should be possible to understand the semantics of a function without considering its dynamic behaviour.*

An *evaluation strategy* is a function that specifies the dynamic behaviour required when computing a value of a given type. A strategy makes no contribution towards the value being computed by the algorithmic component of the function: it is evaluated purely for effect, and hence it returns just the nullary tuple `()`.

```
type Strategy a = a -> ()
```

### 4.3.2 Strategies Controlling Evaluation Degree

The simplest strategies introduce no parallelism: they specify only the evaluation degree. The simplest strategy is termed `r0` and performs no reduction at all. Perhaps surprisingly, this strategy proves very useful, e.g. when evaluating a pair we may want to evaluate only the first element but not the second.

```
r0 :: Strategy a
r0 _ = ()
```

Because reduction to WHNF is the default evaluation degree in GPH, a strategy to reduce a value of any type to WHNF is easily defined:

```
rwhnf :: Strategy a
rwhnf x = x `seq` ()
```

Many expressions can also be reduced to *normal form* (NF), i.e. a form that contains no redexes, by the `rnf` strategy. The `rnf` strategy can be defined over built-in or datatypes, but not over function types or any type incorporating a function type as few reduction engines support the reduction of inner redexes within functions. Rather than defining a new `rnfX` strategy for each data type `X`, it is better to have a single overloaded `rnf` strategy that works on any data type. The obvious solution is to use a Haskell type class, `NFData`, to overload the `rnf` operation. Because NF and WHNF coincide for built-in types such as integers and booleans, the default method for `rnf` is `rwhnf`.

```
class NFData a where
  rnf :: Strategy a
  rnf = rwhnf
```

For each data type an instance of `NFData` must be declared that specifies how to reduce a value of that type to normal form. Such an instance relies on its element types, if any, being in class `NFData`. Consider lists and pairs for example.

```
instance NFData a => NFData [a] where
  rnf [] = ()
  rnf (x:xs) = rnf x `seq` rnf xs

instance (NFData a, NFData b) => NFData (a,b) where
  rnf (x,y) = rnf x `seq` rnf y
```

### 4.3.3 Combining Strategies

Because evaluation strategies are just normal higher-order functions, they can be combined using the full power of the language, e.g. passed as parameters or composed using the function composition operator. Elements of a strategy are combined by sequential or parallel composition (`seq` or `par`). Many useful strategies are higher-order, for example, `seqList` below is a strategy that sequentially applies a strategy to every element of a list, in essence mapping a strategy and then folding the `seq` combinator over the list. For example, the strategy `seqList r0` evaluates just the spine of a list, and `seqList rwhnf` evaluates every element of a list to WHNF. There are analogous functions for every datatype, indeed in Haskell 1.3 and later versions (Peterson et al. 1996) constructor classes can be defined that work on arbitrary datatypes. The strategic examples in this thesis are presented in Haskell 1.2 for pragmatic reasons: they are extracted from programs run on our efficient parallel implementation of Haskell 1.2 (Trinder, Hammond, Mattson Jr., Partridge & Peyton Jones 1996). However, the current version of the strategies module does support Haskell 1.4, too.

```
seqList :: Strategy a -> Strategy [a]
seqList strat []     = ()
seqList strat (x:xs) = strat x `seq` (seqList strat xs)
```

### 4.3.4 Data-Oriented Parallelism

A strategy can specify parallelism and sequencing as well as evaluation degree. Strategies specifying data-oriented parallelism describe the dynamic behaviour in terms of some data structure. For example `parList` is similar to `seqList`, except that it applies the strategy to every element of a list in parallel.

```
parList :: Strategy a -> Strategy [a]
parList strat []     = ()
parList strat (x:xs) = strat x `par` (parList strat xs)
```

Data-oriented strategies are applied by the `using` function which applies the strategy to the data structure `x` before returning it. The expression `` x `using` s `` is a *projection* on `x`, i.e. it is both a retraction (`` x `using` s `` is less defined than `x`) and idempotent (`` (x `using` s) `using` s `` = `` x `using` s ``). The `using` function is defined to have a lower precedence than any other operator because it acts as a separator between algorithmic and behavioural code.

```
using :: a -> Strategy a -> a
using x s = s x `seq` x
```

A strategic version of the parallel map encountered in Section 4.2.2 can be written as follows. Note how the algorithmic code `map f xs` is cleanly separated from the strategy. The `strat` parameter determines the dynamic behaviour of each element of the result list, and hence `parMap` is parametric in some of its dynamic behaviour.

```
parMap :: Strategy b -> (a -> b) -> [a] -> [b]
parMap strat f xs = map f xs `using` parList strat
```

### 4.3.5 Control-Oriented Parallelism

Control-oriented parallelism is typically expressed by a sequence of strategy applications composed with `par` and `seq` that specifies which subexpressions of a function are to be evaluated in parallel, and in what order. The sequence is loosely termed a strategy, and is invoked by either the `demanding` or the `sparking` function. The Haskell `flip` function simply reorders a binary function's parameters.

```
demanding, sparking :: a -> () -> a

demanding = flip seq
sparking  = flip par
```

The control-oriented parallelism of `pfib` can be expressed as follows using `demanding`. The LinSolv and Lolita programs in Sections 4.6 and 4.5 contain more elaborate examples of using `sparking`.

```
pfib n
  | n <= 1    = 1
  | otherwise = (n1+n2+1) `demanding` strategy
    where
      n1 = pfib (n-1)
      n2 = pfib (n-2)
      strategy = rnf n1 `par` rnf n2
```

The control-oriented parallelism of quicksort can be expressed with the following strategy, selecting `losort` and `hisort` for parallel evaluation.

```
quicksortS (x:xs) = losort ++ (x:hisort) `using` strategy
                    where
                      losort = quicksortS [y|y <- xs, y < x]
                      hisort = quicksortS [y|y <- xs, y >= x]
                      strategy result = rnf losort `par`
                                        rnf hisort `par`
                                        rnf result
```

### 4.3.6 Additional Dynamic Behaviour

Strategies can control other aspects of dynamic behaviour, thereby avoiding cluttering the algorithmic code with them. A particularly important example for the scope of this thesis is a thresholding mechanism that controls thread granularity. In `pfib` for example, granularity is improved for many machines if threads are not created when the argument is small. The use of thresholding in Lolita is discussed in Section 4.5.

```
pfibT n
  | n <= 1    = 1
  | otherwise = (n1+n2+1) `demanding` strategy
    where
      n1 = pfibT (n-1)
      n2 = pfibT (n-2)
      strategy = if n > 10
                   then rnf n1 `par` rnf n2
                   else ()
```

Another example of a generic strategy that affects granularity, i.e. the computation costs of potentially parallel threads, is the `parGranList` strategy below. This strategy uses a granularity estimate function and creates the parallelism in an order of decreasing granularity. This strategy has been developed by the author during the performance tuning of a very coarse-grained parallel bowing algorithm (Hall et al. 1997).

```
parGranList :: Strategy a -> (a -> Int) -> [a] -> Strategy [a]
parGranList s gran_estim l_in = \ l_out ->
 parListByIdx s l_out $
 sortedIdx gran_list (sortLe ( \ (i,_) (j,_) -> i>j) gran_list)
 where
  -- spark list elems of l in the order specified by  (i:idxs)
  parListByIdx s l [] = ()
  parListByIdx s l (i:idxs) = parListByIdx s l idxs `sparking` s (l!!i)
  -- get the index of y in the list
  idx y [] = error "idx: x not in l"
  idx y ((x,_):xs) | y==x      = 0
                   | otherwise = (idx y xs)+1
  -- the `schedule' for sparking: list of indices of sorted input list
  sortedIdx l idxs = [ idx x l | (x,_) <- idxs ]
  -- add granularity info to elems of the input list
  gran_list = map (\ l -> (gran_estim l, l)) l_in
```

The purpose of the `parGranList` strategy is to spark all elements in the list `l_out` in an order of decreasing granularity. The function `gran_estim` provides an estimate of the granularity. Note that this estimate has to be applied to the input list `l_in` determining the order of the sparks in the output list. Thus, this strategy abstracts over the concrete definition of how to compute the results in the output list. The strategy proceeds in four steps:

1. First granularity estimates are added to each list element yielding `gran_list`. The construct `\ l -> ...` represents a lambda expression in Haskell, i.e. an anonymous function with the argument `l` and the body ...

2. Then the resulting list is sorted by these estimates using the library function `sortLe`, which takes a predicate, the less-than-or-equal function to be used for sorting, as the first argument.

3. In order to obtain a "schedule" for the order in which the list elements should be sparked, a list of indices of the sorted list is computed using `sortedIdx`.

4. Finally, the index-list is used as the schedule for the `parListByIdx` strategy, which introduces parallelism via a `sparking` clause. The `l!!i` construct is used to extract the `i`-th element from the list `l`.

For clarity, the current version separates the sorting of the list from obtaining the list of indices, yielding a quadratic algorithm. This could be improved further by merging both steps.

Clearly, this strategy encodes a deeper insight into the parallel behaviour of the program than previous strategies. The original motivation for designing this strategy came from the observation that in a coarse-grained program, with largely varying computation times, it is crucial to generate the largest thread first in order to minimise a sequential tail with only the largest thread executing. In a typical process of developing a parallel algorithm the programmer starts with examining the types on the most important data structures and uses pre-defined parallel strategies on these types, e.g. `parList` over list structures. Then, in the performance tuning stage, the programmer might try to improve the behaviour by encoding a particular parallel behaviour in the algorithm as it has been done with the `parGranList` strategy above. The discussion of the LinSolv algorithm in Section 4.6 elaborates this tuning process further.

### 4.3.7 Strategic Function Application

This section discusses one of the author's contributions to the latest version of evaluation strategies as part of his parallelisation of Lolita. The initial version of parallel Lolita was written with `using`-based pipelines. Introducing the notion of strategic function application and rewriting the code in this style simplified the overall structure significantly.

In pipelined parallelism a sequence of stream-processing functions are composed together, each consuming the stream of values constructed by the previous stage and producing a new stream. This kind of parallelism is easily expressed in a non-strict language by function composition. The non-strict semantics ensures that no barrier synchronisation is required between the different stages.

When using strategies to describe this kind of parallelism a function composition is needed, which applies a strategy to the intermediate value. Based on this observation *strategic function application* and *strategic function composition* are introduced. The new operators correspond to function application `$` and function composition `.` defined in the Haskell prelude. The strategic function application takes one additional argument, a strategy `s`, which is applied to the argument. The parallel version of the operator, `$||`, applies the strategy and the function in parallel, thereby overlapping two stages in the pipeline. The sequential version of this operator, `$|`, first applies the strategy and then the function to the argument. This introduces a synchronisation barrier and may be used to define evaluation order. However, the strategy may itself define parallelism, e.g. over the structure of the argument. The `.||` and `.|` operators define the same behaviour for function compositions. The definition of these operators in GPH is given below.

```
infixl 6 $|, $||       -- strategic function application
infixl 9 .|, .||       -- strategic function composition

($|), ($||) :: (a -> b) -> Strategy a -> a -> b
(.|), (.||) :: (b -> c) -> Strategy b -> (a -> b) -> (a -> c)

($|) f s x  = f x `demanding` s x
($||) f s x = f x `sparking`  s x

(.|) f s g = \ x -> let  gx = g x
                    in   f gx `demanding` s gx
(.||) f s g = \ x -> let  gx = g x
                     in   f gx `sparking` s gx
```

An often used example of the modularity of functional languages is the definition of the sum-of-squares function for computing the sum of the first $n$ integer values via the composition of three separate functions. With the new construct of strategic function application we can define a parallel behaviour of the same definition in a very natural way without obscuring the original algorithmic code:

```
sum_of_squares :: Int -> Int
sum_of_squares n = sum            $|| parList rnf $   -- [Int]
                   map (^2)       $|| rnf $           -- [Int]
                   enumFromTo 1 n
```

The functions are applied via the parallel `$||` operator to obtain a parallel pipeline structure. Furthermore, the types of the intermediate lists, `[Int]`, already suggest a strategy for exposing additional data parallelism in the code: `parList rnf`. However, in this case we have chosen not to use the parallelism over the list generated by the `enumFromTo` library function, because it contains too little computation for each of the list elements. As a result, this function defines a pipeline strategy with data parallelism over one of the two intermediate list structures (see Figure 4.1). It is easy to experiment with the parallelism in the code, e.g. by merging pipeline stages, which amounts to replacing `$||` with a `$|` operator. The data parallelism over the intermediate data structures can be simply modified by choosing different strategies as arguments to the `$||` operator. Because none of these changes require to examine the code for the function `sum`, `map`, and `enumFromTo`, this example shows how the modularity, obtained in functional languages via non-strict data structures and

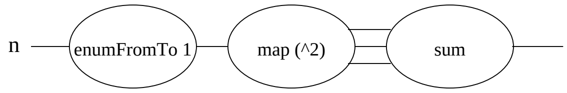


**Figure 4.1** Structure of sum-of-squares

function composition, carries over to the definition of the parallel behaviour of the code.

As a comparison of two versions of a more sophisticated strategy we now discuss the back end in the Lolita system, which interprets semantic information obtained in a previous analysis in the system. This comparison illustrates that keeping intermediate values anonymous increases the readability of the program significantly. A `using`-based version of the back end in Lolita can be written as follows. Details of the code will be discussed in Section 4.5.

```
back_end inp opts
 = r8 `demanding` strat
   where
     r1 = unpackTrees inp
     r2 = unifySameEvents opts r1
     r3 = storeCategoriseInformation r2
     r4 = unifyBySurfaceString r3
     r5 = addTitleTextrefs r4
     r6 = traceSemWhole r5
     r7 = optQueryResponse opts r6
     r8 = mkWholeTextAnalysis r7
     strat = (parPair rwhnf (parList rwhnf)) inp                 `seq`
             (parPair rwhnf (parList (parPair rwhnf rwhnf))) r1  `seq`
             rnf r2                                              `par`
             rnf r3                                              `par`
             rnf r4                                              `par`
             rnf r5                                              `par`
             rnf r6                                              `par`
             (parTriple rwhnf (parList rwhnf) rwhnf) r7          `seq`
             ()
```

By using strategic function application the same code can be written more succinctly as follows. The separation of algorithmic and behavioural code is maintained by allowing strategies only as arguments to the strategic function application.

```
back_end inp opts =
  mkWholeTextAnalysis    $|   parTriple rwhnf (parList rwhnf) rwhnf $
  optQueryResponse opts  $|| rnf $
  traceSemWhole          $|| rnf $
  addTitleTextrefs       $|| rnf $
  unifyBySurfaceString   $|| rnf $
  storeCategoriseInf     $|| rnf $
  unifySameEvents opts   $|   parPair rwhnf (parList (parPair rwhnf rwhnf)) $
  unpackTrees            $|   parPair rwhnf (parList rwhnf)  $
  inp
```

Strategic function application has proven useful in particular for the parallelisation of Lolita (see Section 4.5.3). The Alpha-Beta search algorithm described in Section 4.4 has a top-level pipeline structure. However, in this case there is far less potential parallelism in the pipeline structure.

The importance of strategic function application and composition for parallel programming is underlined by the fact that function composition is considered the basic building block for constructing large programs from independent modules (Hughes 1989). The software engineering advantages, such as improved modularity, for sequential program development are well known. In the parallel setting strategic function composition also facilitates a data-oriented approach to parallelisation, making use of the modularity provided by lazy languages.

## 4.4 Alpha-Beta Search

The first example program is the Alpha-Beta search algorithm, typical of artificial intelligence applications. It is mainly used for game-playing programs to find the best next move by generating all possible moves up to a certain depth, applying a static evaluation function to each of the leaves in this search tree, and combining the result by picking the best move for the player assuming that the opponent picks the worst move for the player. In a more general setting this algorithm can be used for heuristic search. The idea of the heuristics is that the quality of the result depends on the static evaluation function as well as on the search depth. If the latter is sufficiently high a very simple static evaluation function can be used.

This section discusses two versions of the Alpha-Beta search algorithm: a *simple* version, and a *pruning* version. Both versions are based on the Miranda[1] code presented

[1]Miranda is a trademark of Research Software Ltd.

by Hughes (1989) in order to demonstrate the strengths of lazy functional languages. Based on the generic Alpha-Beta search algorithm two simple games (tic-tac-toe and escape) have been implemented. An interesting aspect of this algorithm is the fact that the pruning version relies on laziness to prune the search tree based on intermediate results of the computation. This behaviour is crucial for the efficiency of the sequential algorithm, and has to be preserved in the parallel algorithm.

This section presents both parallel versions and studies their parallel runtime behaviours. The parallel algorithms show how the use of strategies allows the programmer to develop an efficient parallel algorithm without sacrificing the advantages of the original lazy algorithm, namely its modularity and efficiency. A description of both algorithms and a comparison of the parallelisation with that of other applications is given in Loidl & Trinder (1997).

### 4.4.1 Simple Algorithm

In the simple algorithm each possible next move is evaluated independently yielding a divide-and-conquer structure of the algorithm. The result is either the maximum, player's move, or the minimum, opponent's move, of the evaluations of these positions. As discussed by Hughes (1989) this algorithm can be very naturally derived as a sequence of function compositions (see Figure 4.2). The stages in the pipeline perform the following tasks:

1. Construct a tree with positions as nodes and all possible next moves as subtrees. This is done by repeatedly applying a `newPosition` function to the nodes in the tree, alternating between the functions for the two players, `repTree`.

2. Prune the tree, which might be infinite at this stage, to a fixed depth to bound the search via `prune`. The search depth is an argument to the algorithm.

3. Map a static evaluation function over all nodes of the tree via `mapTree`.

4. Crop off subtrees from winning or losing positions via `cropTree`. If such a position is found it is not necessary to search deeper in a subtree.

5. Finally, pick the maximum, or minimum, of the resulting evaluations in order to determine the value of the current position via `mise f g`. The functions `f`

```
bestMove :: Int -> Piece -> Player -> Player -> Board -> Evaluation
bestMove depth p f g = (mise f g) .
                       cropTree .
                       (mapTree (static p)) .
                       (prune depth) .
                       repTree (newPositions p)
                               (newPositions (opposite p))
```

**Figure 4.2** Top level structure of choosing the best next move

and g represent the combination functions for the two players, maximum or minimum respectively, and alternate when traversing the tree.

### Dynamic Behaviour

The fact that the results in all subtrees can be computed independently makes parallelisation rather easy. For both versions of the algorithm the following four sources of parallelism can be used.

**Top Level Pipeline.** An obvious approach to parallelise this algorithm is to use pipeline parallelism between the stages of the pipeline. However, it is crucial not to force the intermediate values too far. In particular, the result of the `repTree` stage might be an infinite tree.

**Parallel Static Evaluation Function.** The idea of a parallel static evaluation function is to reduce the costs of the function, which will be mapped over the leaves of the pruned search tree. This only makes sense for a rather time consuming static evaluation function, otherwise it creates a lot of fine-grained parallelism. However, an underlying assumption of the Alpha-Beta search algorithm is that the static evaluation function can be very simple when using a tree search structure to determine the best value. In the example implementations, the static evaluation function computes the distance of the current position to a set of known winning positions. The parallel version computes all distances in parallel.

**Parallel Higher-Order Functions over Trees.** Parallelising the definitions of some higher-order functions is a bottom-up approach. It can be used for the parallelisation of many functional programs. In this case a parallel version of a map function over search trees, `mapTree`, is used. However, the measurements in Table 4.1 show, that without any knowledge about the context in which these higher-order functions are used a lot of redundant work may be generated resulting in extremely poor parallelism.

**Data Parallelism over all Possible Next Moves.** In a data parallel approach the goal is to evaluate all possible next moves in parallel. It is a top-down approach and turns out to be the best source of parallelism in particular for an algorithm with no dependencies between the evaluations of the subtrees. A simple `parMap rnf` strategy can be used to capture the dynamic behaviour of this function. The only necessary change in the algorithm affects the `mise` function in Stage 5 of the algorithm, shown in Figure 4.3. This function takes the two combination functions, either the binary `max` or `min` function, and a tree of static evaluations of positions in the game, as arguments. It then recursively maps the `mise` function over all subtrees, switching the functions `f` and `g` to record the switch of turns. Finally, the combination function at the current level, `f`, is folded to obtain the score of the current position.

```
-- This does simple minimaxing without pruning subtrees based on
-- intermediate evaluations (i.e. purely compositional)
mise :: Player -> Player -> (Tree Evaluation) -> Evaluation
mise f g (Branch a []) = a
mise f g (Branch _ l) = foldr f (g OWin XWin) (parMap rnf (mise g f) l)
```

**Figure 4.3** Data parallel combination function in the simple Alpha-Beta search algorithm

### Performance Measurements

The measurements of both versions of the algorithm under the GranSim simulator are summarised in Table 4.1. The setup used in these measurements models a shared memory machine with 32 processors, a latency of 64 machine cycles, and bulk fetching. The first four data columns of this table show the results of the simple algorithm

**Table 4.1** Measurements of the simple and the pruning Alpha-Beta search algorithm

| | Simple Algorithm | | | | Pruning Algorithm | | | |
|---|---|---|---|---|---|---|---|---|
| | Runtime (kcycles) | Avg Par | Total Work | SpdUp | Runtime (kcycles) | Avg Par | Total Work | SpdUp |
| *Position I* | (standard) | | | | | | | |
| Sequential | 60,297 | | | | 34,363 | | | (1.75) |
| Par Pipeline | 60,297 | 1.0 | 100% | 1.00 | 34,370 | 1.0 | 100% | 0.99 |
| Par Static Eval | 21,091 | 3.1 | 108% | 2.85 | 12,099 | 3.1 | 109% | 2.84 |
| Data Par | 3,503 | 26.4 | 153% | 17.21 | 2,265 | 23.7 | 156% | 15.17 |
| Par h.o. fcts | 4,954 | 20.9 | 172% | 12.16 | 4,248 | 24.2 | 299% | 8.08 |
| Par Static Eval & Data Par | 3,507 | 28.5 | 166% | 17.19 | 2,156 | 27.6 | 173% | 15.93 |
| Par h.o. fcts & Data Par | 3,701 | 28.2 | 173% | 16.29 | 3,683 | 28.3 | 303% | 9.32 |
| *Position II* | (early solution) | | | | | | | |
| Sequential | 4,427 | | | | 4,703 | | | (0.94) |
| Par Pipeline | 4,427 | 1.0 | 100% | 1.00 | 4,706 | 1.0 | 100% | 0.99 |
| Par Static Eval | 1,772 | 2.9 | 116% | 2.49 | 1,898 | 2.9 | 117% | 2.47 |
| Data Par | 1,152 | 13.9 | 362% | 3.84 | 1,075 | 13.1 | 299% | 4.37 |
| Par h.o. fcts | 759 | 9.6 | 165% | 5.83 | 811 | 9.0 | 155% | 5.79 |
| Par Static Eval & Data Par | 775 | 23.2 | 406% | 5.71 | 779 | 20.4 | 338% | 6.03 |
| Par h.o. fcts & Data Par | 919 | 20.4 | 424% | 4.81 | 1,001 | 18.9 | 403% | 4.69 |
| *Position III* | (large search tree) | | | | | | | |
| Sequential | 145,720 | | | | 90,377 | | | (1.61) |
| Par Pipeline | 145,720 | 1.0 | 100% | 1.00 | 90,385 | 1.0 | 100% | 0.99 |
| Par Static Eval | 48,808 | 3.3 | 111% | 2.98 | 29,891 | 3.3 | 109% | 3.02 |
| Data Par | 6,621 | 29.1 | 132% | 22.00 | 7,699 | 16.2 | 138% | 11.73 |
| Par h.o. fcts | 9,345 | 21.4 | 137% | 15.59 | 8,093 | 24.6 | 220% | 11.16 |
| Par Static Eval & Data Par | 7,083 | 29.3 | 142% | 20.57 | 5,210 | 25.7 | 148% | 17.34 |
| Par h.o. fcts & Data Par | 6,882 | 29.3 | 138% | 21.17 | 6,802 | 29.6 | 223% | 13.28 |

when using the different sources of parallelism. All runtimes are given in machine-independent kilocycles. The total work column measures the total work compared to a sequential run and is therefore a measure of the redundant work, in particular of speculative parallelism. The three horizontal sections in the table represent three different positions that have been analysed: a standard opening position (I) with a sequential runtime of 60,297 kilocycles; a winning position (II) with a sequential runtime of 4,427 kilocycles; and a position generating a large search tree (III) with a sequential runtime of 145,720 kilocycles.

The parallel *pipeline* version creates hardly any parallelism at all. This is due to the fact that it is not possible to force the search tree before pruning it without generating a huge amount of redundant work. This result differs significantly from the results with programs like Lolita, where the top-level structure of the whole algorithm is a parallel pipeline. The *parallel static evaluation function* generates conservative parallelism shown by the small amount of total work performed. However, the degree of parallelism is rather small: in this example program the distance of the current position to a small set of winning positions is computed in a data parallel fashion. Another disadvantage is the fine-grained nature of the parallelism, i.e. each of the generated threads performs very little computation. The *data parallelism* over all next positions proves to be the best source of parallelism. The simple algorithm will only cut-off subtrees if it finds a winning position in one of the subtrees. Therefore, this data parallelism is conservative except for the case where a winning position is found as in Position II. Note that in the latter case the simple sequential algorithm performs even better than the pruning algorithm indicated by the algorithm speedup of 0.94, in brackets, in the last column. Finally, the *higher-order functions* approach generates the largest amount of redundant work shown by the high total work percentage. Here a parallel tree map of the static evaluation function is used. However, this also maps the evaluation function on nodes that are actually pruned in the sequential algorithm. Combining data parallelism with parallel static evaluation does not improve the performance in general. Although the average parallelism increases, the speedup actually drops for Positions I and III because the additional parallelism is very fine-grained.

For the simple Alpha-Beta algorithm using only data parallelism gives an almost perfect utilisation of the machine, provided that the search space is large enough. If a solution is found early on then the speedup will naturally drop (see Position II in Table 4.1). However, for more realistic games than tic-tac-toe the search space should easily be large enough because of the exponential growth of the search tree.

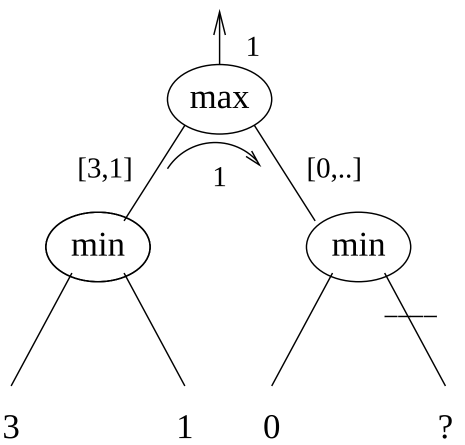


**Figure 4.4** Pruning subtrees in the optimised Alpha-Beta search algorithm

### 4.4.2 Pruning Algorithm

The simple algorithm described in the previous section lacks one crucial optimisation of the Alpha-Beta search: the pruning of subtrees based on intermediate results. The pruning algorithm returns an increasing list (player's move) of approximations with the exact value as last list element rather than a single value. The main pruning function, `minleq`, has to test whether the opponent's move from a subtree can be ignored (see Figure 4.4). This is the case if the worst result of the decreasing list $l$, i.e. its minimum, is no better, i.e. less than or equal to, the intermediate result $x$. Or more formally: minimum $l \leq x \Leftrightarrow$: `minleq` $l$ $x$. Since `minleq` works on decreasing lists it can stop examining the list as soon as it finds a value less than $x$. Thus, laziness is used to ignore parts of the list of approximations, which amounts to pruning subtrees in the search tree. A complete description of this lazy functional pruning algorithm can be found in Hughes (1989).

In the sequential code in Figure 4.5 the prelude functions `min` and `max` from the simple algorithm are replaced with functions `min'` and `max'`, respectively. The new functions operate over lists of approximations. In implementing the behaviour described in the previous paragraph the `betterthan` function will stop examining list elements of `next` when it is clear that the final result will not be better than the value `a` found so far. Figure 4.4 illustrates this behaviour. After having determined the value of the left subtree and the value 0 in the right subtree it is not necessary to examine the

```
-- A pruning version of alpha-beta search
mise :: Player -> Player -> (Tree Evaluation) -> [Evaluation]
mise f g (Branch a []) = [a]
mise f g (Branch _ l) = f (map (mise g f) l)

betterthan :: (Evaluation -> [Evaluation] -> Bool) -> -- maxleq or minleq
              ([Evaluation] -> Evaluation) ->         -- max' or min'
              Evaluation ->                           -- Score to compare
              [[Evaluation]] ->                       -- list of approxs
              [Evaluation]
betterthan _ _ _ [] = []
betterthan better_than_worst worst a (next:rest)
 | a `better_than_worst` next = betterthan better_than_worst worst a rest
 | otherwise              = m : betterthan better_than_worst worst m rest
                                where m = worst next

-- minleq y l <=> minimum l <= y
minleq :: Evaluation -> [Evaluation] -> Bool
minleq y []      = False
minleq y (x:xs)
  | x <= y    = True       -- throws away the rest of the list!
  | otherwise = minleq y xs

-- used as argument to mise
max' :: [[Evaluation]] -> [Evaluation]
max' (first:rest) = m : betterthan minleq minimum m rest
                    where m = minimum first  -- strict in first
```

**Figure 4.5** Pruning version of the Alpha-Beta search

rightmost leaf. The overall maximum is guaranteed to be at least 1.

### Dynamic Behaviour

Unfortunately, the pruning version seriously complicates the parallelisation of the algorithm. We have already seen in the simple algorithm that the most promising source of parallelism is the parallel evaluation of all next positions. However, using a simple `parList rnf` strategy over all next positions is no longer advisable, since this might result in a lot of redundant work, if many subtrees can be pruned. The measurements of the data parallel strategy on the pruning algorithm in Table 4.1 show a rather high degree of redundant work. In fact, in the data parallel strategy on Position III the parallel simple version is even faster than the highly speculative parallel pruning version of the algorithm!

A better approach for parallelisation is to force only an initial segment in the list of

```
-- Parallel version of the pruning version
mise :: Player -> Player -> (Tree Evaluation) -> [Evaluation]
mise f g (Branch a []) = [a]
mise f g (Branch _ l) =
  f
    -- force the first n elements of the result list
    ((map (mise g f) l)
     `using` \ xs -> if force_len==-1  -- infinity
                       then parList rnf xs `par` ()
                       else parList rnf (take force_len xs)    `par`
                            parList rwhnf (drop force_len xs)  `par`
                                                               ()
    )
```

**Figure 4.6** Strategy for a parallel pruning version with a static force length

possible next positions. We call the length of this segment the "force length". We have experimented with static force lengths as well as dynamic force lengths that depend on the level in the search tree. To date the best results have been obtained from using a static force length as shown in the parallel code for `mise` in Figure 4.6. The algorithmic code for `mise` is unchanged compared to the sequential version. The strategy uses a global constant `force_len` to determine how much of the list `xs` should be evaluated. Because strategies are simply Haskell functions, the prelude functions `take` and `drop` can be used for that purpose. Note that the force length represents a trade-off between increasing the degree of parallelism and reducing the total amount of work being done.

### Performance Measurements

Figure 4.7 compares the speedups of the pruning version of Alpha-Beta search under GranSim, using the same setup as in the previous measurements. The x-axis shows the static force length, the y-axis the speedup. The left hand graph uses a program implementing tic-tac-toe, the right hand graph uses an implementation of a similar game, escape, with a search space of comparable size but asymmetric winning conditions.

The left hand graph shows for the data parallel strategy a large improvement when increasing the force length, in particular for Position III. A purely conservative data parallel strategy (i.e. the force length is 0) achieves a speedup of only 8.58 because the amount of available parallelism drops early on in the computation (see Figure 4.8).

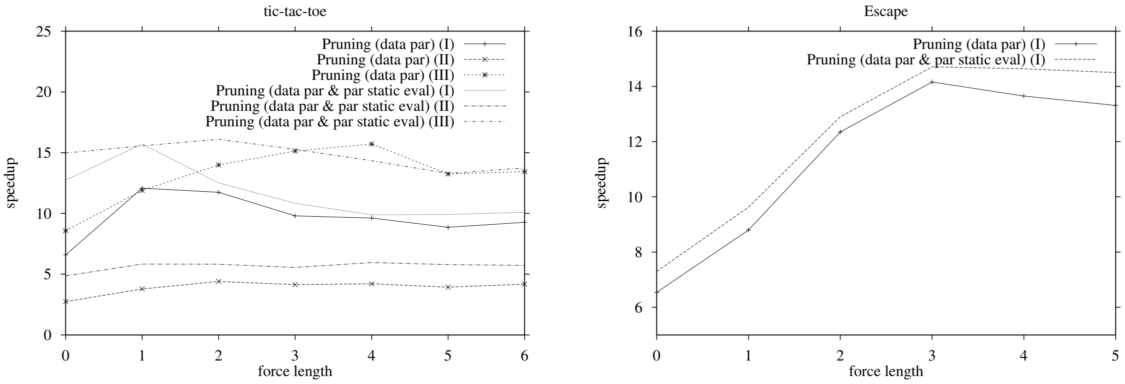


**Figure 4.7** Speedup with varying force length (GranSim)

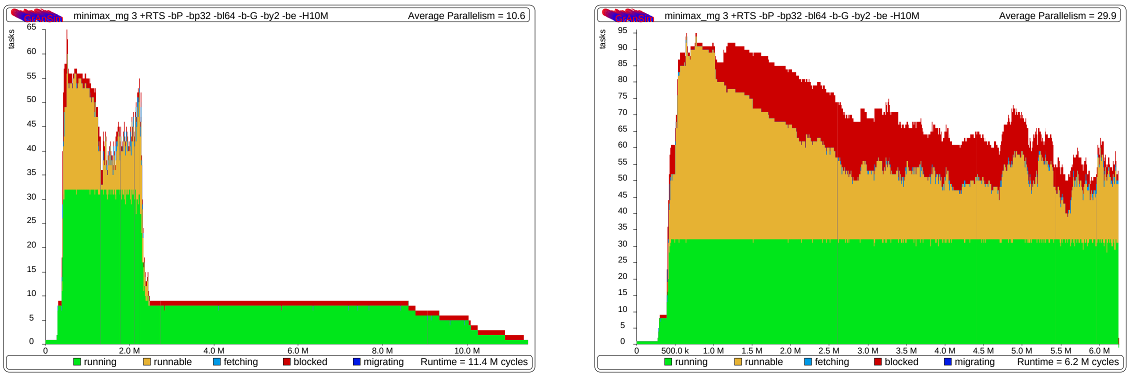


**Figure 4.8** Data parallel versions with static force lengths of 0 and 4

In contrast, with a force length of 4 the speedup is 15.71. After that the percentage of redundant work done in the parallel algorithm increases too much to achieve a further improvement. For Position II, which finds a winning position early on in the search, parallelism can achieve hardly any improvement because almost all potential parallelism in the algorithm is pruned. The versions additionally using a parallel static evaluation function usually outperform the versions with data parallelism alone, because the small amount of conservative parallelism in the static evaluation can make use of idle time on the machine. This is in contrast to the simple algorithm, where the data parallel evaluation function generates enough parallelism to keep the machine busy. This can be seen in Table 4.1, comparing the speedups of the lines for data parallelism and data parallelism together with a parallel static evaluation function.

## 4.5 Lolita

### 4.5.1 Algorithm

The Lolita natural language engineering system (Morgan et al. 1994) has been developed at the University of Durham over several years. It has not originally been written with a parallel execution of the code in mind. The team's interest in parallelism is partly as a means of reducing runtime, and partly also as a means to increase functionality within an acceptable response-time. The overall structure of the program bears some resemblance to that of a compiler, being formed from the following large stages:

- Morphology (combining symbols into tokens; similar to lexical analysis);
- Syntactic Parsing (similar to parsing in a compiler);
- Normalisation (to bring sentences into some kind of normal form);
- Semantic Analysis (compositional analysis of meaning);
- Pragmatic Analysis (using contextual information from previous sentences).

These stages form the core of Lolita. Depending on how Lolita is to be used, a final additional stage may perform a discourse analysis, the generation of text (e.g. in a translation system), or it may perform inference on the text to answer queries. This design of the system yields a very flexible and modular structure. A more detailed discussion of the Lolita system and of its parallelisation is given in Loidl et al. (1997). The parallelisation has been done as joint work with the group at the University of Durham.

Central to Lolita's flexibility is the semantic network, a graph based knowledge representation used in the core of Lolita. In the semantic network concepts and relationships are represented by nodes and arcs respectively, with knowledge being extracted by graph traversal. The task of the analysis stages is to transform the possibly ambiguous input into a sub-graph of the semantic network. Application-dependent backend stages can then extract pieces of the semantic network and present it in the required form.

### 4.5.2 Sequential Profiling

As a preparation for parallelising such a large program the author has performed sequential profiling of the code. This did not reveal a particular hotspot in the program although the syntactic parsing stage is the biggest component in the top-level structure with about 20% of the execution time. However, this stage makes heavy use of C-functions, called from within Haskell, to optimise the time consuming parsing process. This complicates a parallelisation of the parsing stage. The Haskell part of the parsing, however, can be parallelised without major recoding.

### 4.5.3 Top Level Pipeline

Without a clear hotspot in the sequential execution of the program a pipeline approach is a promising way to achieve enough parallelism for a four processor shared-memory machine such as a Sun SPARCserver. The structure of a pipeline parallel version is shown in Figure 4.9. Each stage listed above is executed by a separate thread, which are linked to form a pipeline. Note that in order to make use of the multi-threaded runtime-system, which overlaps computation and communication, the parallel algorithm should contain more threads than there are processors available. The key step in parallelising the system is to define strategies on the complex intermediate data structures, e.g. parse trees, that are used to communicate between these stages. This data-oriented approach simplifies the top-down parallelisation of this very large system, since it is possible to define the parallelism over parts of a data structure without considering the algorithms that produce that data structure. This approach hides unnecessary information about the generation of the data structure and is in the spirit of functional programming, which tries to achieve modularity by composing flexible, possibly higher-order, functions.

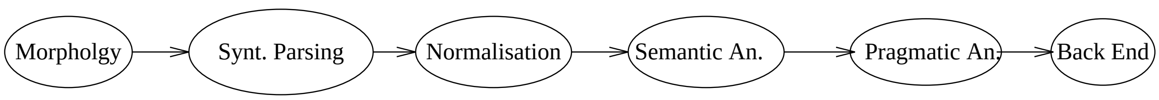


**Figure 4.9** Overall pipeline structure of Lolita

The code of the top-level function `wholeTextAnalysis` in Figure 4.10 uses strategic function application as the basic operator to introduce parallelism (see Section 4.3.7).

The algorithm is separated from the dynamic behaviour in each stage by using the `$||` operator. In a first parallel version the same separation has been achieved with an explicit pipeline strategy. However, this required to name every intermediate value in the pipeline. As a result many additional variables had to be added to the code, obscuring the algorithmic part of the code. This experience was the main motivation for developing the strategic function application operator.

Note that this code uses a `parList` strategy in the definition of `rawParseForest` in the parsing stage to describe data parallelism over the whole input by processing sentences in the input text in parallel. In the current version of the system it is not possible to use this source of parallelism because the C code in this stage is not re-entrant. Changing the C code to exploit this form of parallelism is ongoing work. The strategies in the individual stages of Figure 4.10 will be discussed in the subsequent sections.

The semantic and pragmatic analysis stages are wrapped into a timeout function in order to guarantee a worst case response time of the system. This indicates that these stages can be very computationally intensive. Therefore, both analyses are kept rather simple in the sequential system. By providing the strategy `evalScores`, in `parse2prag`, speculative parallelism is defined, which allows the system to perform a more sophisticated analysis by examining several possible parse trees. The goal of this strategy is therefore to improve the quality of the result. Section 4.5.5 discusses this issue in more detail. In general, it would be very desirable to improve the quality of semantic and pragmatic analysis in the system. Parallelism inside these stages could be used to maintain good performance despite the increased complexity of the system.

### 4.5.4 Parallel Parsing

One major source of parallelism in the time consuming syntactic parsing stage is the merging of possible parse trees in order to build a parse tree for a whole sentence. One complication in the parsing of natural languages is their ambiguity. Because of this ambiguity the parsing stage produces not just one but a list of possible parse trees. Internally, however, the result is represented as a single tree, which at some points contains alternatives ("or-nodes") representing different possible parses of the subtrees. A lazy function is used to convert this single tree into a list of possible parse

```
wholeTextAnalysis opts inp global =
  result
  where
    -- (1) Morphology
    (g2, sgml) = prepareSGML inp global
    sentences  = selectEntitiesToAnalyse global sgml

    -- (2) Parsing
    rawParseForest = map (heuristic_parse global) sentences
                     `using` parList rnf

    -- (3)-(5) Analysis
    anlys = stateMap_TimeOut (parse2prag opts) rawParseForest global2

    -- (6) Back End
    result = back_end anlys opts

-- Pick the parse tree with the best score from the results of
-- the semantic and pragmatic analysis.  This is done speculatively!

parse2prag opts parse_forest global =
 pickBestAnalysis global  $|| evalScores  $
 take (getParsesToAnalyse global)         $
 map analyse parse_forest
 where
   analyse pt =   mergePragSentences opts $ evalAnalysis
   evalAnalysis = stateMap_TimeOut analyseSemPrag pt global
   evalScores =   parList (parPair rwhnf (parTriple rnf rwhnf rwhnf))

-- Pipeline the semantic and pragmatic analyses
analyseSemPrag parse global =
 prag_transform           $|| rnf   $
 pragm                    $|| rnf   $
 sem_transform            $|| rnf   $
 sem (g,[])               $|| rnf   $
 addTextrefs global       $|  rwhnf $
 subtrTrace global parse

back_end inp opts =
 mkWholeTextAnalysis   $|  parTriple rwhnf (parList rwhnf) rwhnf $
 optQueryResponse opts $|| rnf $
 traceSemWhole         $|| rnf $
 addTitleTextrefs      $|| rnf $
 unifyBySurfaceString  $|| rnf $
 storeCategoriseInf    $|| rnf $
 unifySameEvents opts  $|  parPair rwhnf (parList (parPair rwhnf rwhnf)) $
 unpackTrees           $|  parPair rwhnf (parList rwhnf)  $
 inp
```

**Figure 4.10** The top level function of Lolita

trees. In each or-node the parser, which returns a list of parse trees, must merge the lists of parse trees produced by the recursive calls. In merging these lists the possible parse trees have to be sorted based on some simple syntactic criteria representing the likelihood of a parse, and the laziness of Haskell is crucial. In order to produce one parse tree in an or-node it is only necessary to evaluate the first element in the lists

```
mergeStrategy :: (NFData a, NFData b) =>
                 (ParseForest,FeatureForests) -> Span -> MergeStrategy a b

mergeStrategy (pf,ff) span
  | totalSpan == 0              = MStrat serialMerge
  | percentSpanned >= minSpan   = MStrat parallelMerge
  | otherwise                   = MStrat serialMerge
    where
      percentSpanned = (span * 100) `div` totalSpan
      totalSpan = forestSpan pf
      minSpan = getParsingParPercent (forestGlobal pf)

      parallelMerge :: (NFData a, NFData b) =>
                       [(a,b)]  -> [(a,b)] -> Strategy [(a,b)]
      parallelMerge as bs _
       = fstPairFstList bs `par`
         fstPairFstList as `seq`
         ()

      fstPairFstList :: (NFData a, NFData b) => Strategy [(a,b)]
      fstPairFstList = seqListN 1 (seqPair rwhnf r0)

      serialMerge :: (NFData a, NFData b) =>
                     [(a,b)]  -> [(a,b)] -> Strategy [(a,b)]
      serialMerge as bs
       = r0
```

**Figure 4.11** A granularity control strategy used in the parsing stage

produced by all alternatives.

From a parallelism point of view this behaviour explains why it is not possible to force the evaluation of parts of the parse forest without risking to introduce a high degree of redundant work. Within the parsing process the merging of lists triggers the evaluation of sublists, in particular the evaluation of the quality of possible parses. Although the merging itself is very cheap it triggers work that can be usefully done in parallel.

In order to improve the granularity of the threads produced by the parallel tree traversal in the parsing stage, we apply a thresholding strategy, shown in Figure 4.11, to the "span" in the tree. The span value, which is attached to each node in the tree, specifies the number of leaves in the current subtree. The threshold for generating a parallel process in order to merge all possible subtrees is specified as a percentage of leaves that can be reached from the current node, and this percentage is part of the global system environment. Checking the threshold is very cheap because it only involves the comparison of the `span` argument, as a percentage, with a system parameter assigned to `minSpan`.

The two parallel calls to `fstPairFstList` in `parallelMerge` define parallelism in this stage. Only the first element of the pair is evaluated because it contains the value determining the quality of the resulting parse tree. Thus, the `fstPairFstList` strategy specifies an evaluation degree that is sufficient to select the tree to return as the result of the syntactic parsing stage but without evaluating the tree itself more than necessary.

One strength of strategies is their reusability for different algorithmic code that has the same dynamic behaviour. We were able to exploit this feature with `mergeStrategy` in Figure 4.11 by applying the same polymorphic thresholding strategy to two lists of different types within the syntactic parsing stage. This reuse is highlighted by the parameterisation of the `MergeStrategy` datatype over the two possible types in the list. Both instances of applying `mergeStrategy` are in sub-functions of `heuristic_parse` in Figure 4.10.

The measurements discussed in this section have been performed with GranSim in a setup that models the four processor shared-memory Sun SPARCServer available at Durham. The goal of these measurements is to determine the best value for the span in the `mergeStrategy`. Figure 4.12 shows the activity profiles for Lolita using a span threshold of 50%, left hand graph, and 90%, right hand graph. Both profiles show a good utilisation of the system during the syntactic parsing stage. However, in the left hand graph almost 100 blocked threads and a high number of runnable threads are generated, too. These impose significant runtime overhead in the system. The granularity profile at the left hand side of Figure 4.13 reveals that most of the threads are very fine-grained: 3,422 of the 5,122 threads (67%) are shorter than 2,000 cycles. This leads to a bad ratio of computation versus parallelism overhead.

In comparison, when increasing the span threshold to 90% the number of blocked and runnable threads is reduced significantly (at most 36), and the number of small threads drops drastically, as shown in the right hand graph of Figure 4.12 (note the different scaling in both graphs). Now, only 67 of the 165 threads are shorter than 2,000 cycles (40%). Corresponding to this drop in the total number of threads, especially fine-grained threads, the runtime drops from 754,687 kilocycles in the previous version to 526,842 kilocycles in this version. As a result of these measurements and considering the low amount of parallelism that is required to fully utilise the four processor shared-memory machine, span thresholds around 90% are used for GUM

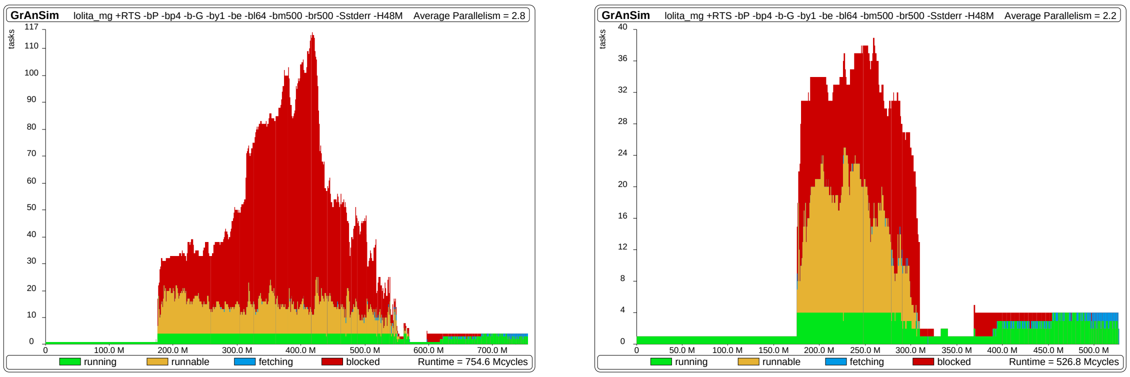


**Figure 4.12** Activity profiles of Lolita with span thresholds of 50% and 90%

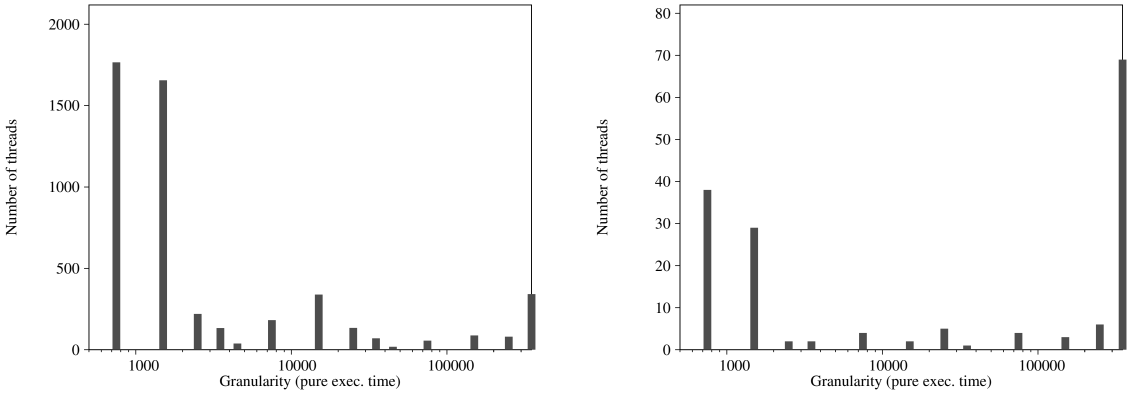


**Figure 4.13** Granularity profiles of Lolita with span thresholds of 50% and 90%

executions of Lolita.

## 4.5.5 Parallel Semantic Analysis

Another source of parallelism can be used to improve the quality of the analysis by applying the semantic and pragmatic analyses in a data-parallel fashion on different possible parse trees for the same sentence. Because of the complexity of these analyses, the sequential system always picks the first parse tree, which may cause the analysis to fail, although it would succeed for a different parse tree. In this case the system cannot produce a result for the current sentence in a sequential setup. Therefore, parallelism in this stage would not reduce the runtime of the system, but might improve the quality of the result.

This additional data parallelism is defined by the strategy `evalScores` in the function `parse2prag` (see Figure 4.10). The parse forest `rawParseForest` contains all possible parses of a sentence. The semantic and pragmatic analyses are then applied to a predefined number, specified in `global`, of these parses. The data parallel strategy `evalScores` is applied to the list of these results and demands only the score of each analysis, the first element in the triple, in order to avoid unnecessary computation at this stage. This score is used in `pickBestAnalysis` to decide which of the parses to choose as the result of the whole text analysis.

The improvements in the quality of the result by analysing several possible parse trees have not been systematically measured, yet. However, considering that about 70% of all sentences that are analysed have several possible parse trees, the possibility to analyse several of them without large additional costs is very attractive from a natural language engineering point of view.

### 4.5.6 Overall Parallel Structure

Figure 4.14 summarises the overall parallel structure arising when all of the sources of parallelism described above are used. The possible data parallelism over the input is depicted by analysing three sentences in parallel in this picture. Note that the number of possible parse trees for the input sentences varies. The syntactic parsing stage is internally parallelised using the granularity control strategy shown in Figure 4.11. Note that the analyses may add nodes to the semantic net. This creates an additional dependence between different instances of the analysis, which is indicated as vertical arcs. Lazy evaluation ensures that this does not completely sequentialise the analyses, however.

It should be emphasised that specifying the strategies that describe this parallel behaviour entailed understanding and modifying only two of about three hundred modules in Lolita and three of the thirty six functions in that module. Apart from the top level function, the only sub-module that has been parallelised is the syntactic parsing stage. If it proves necessary to expose more parallelism it would be

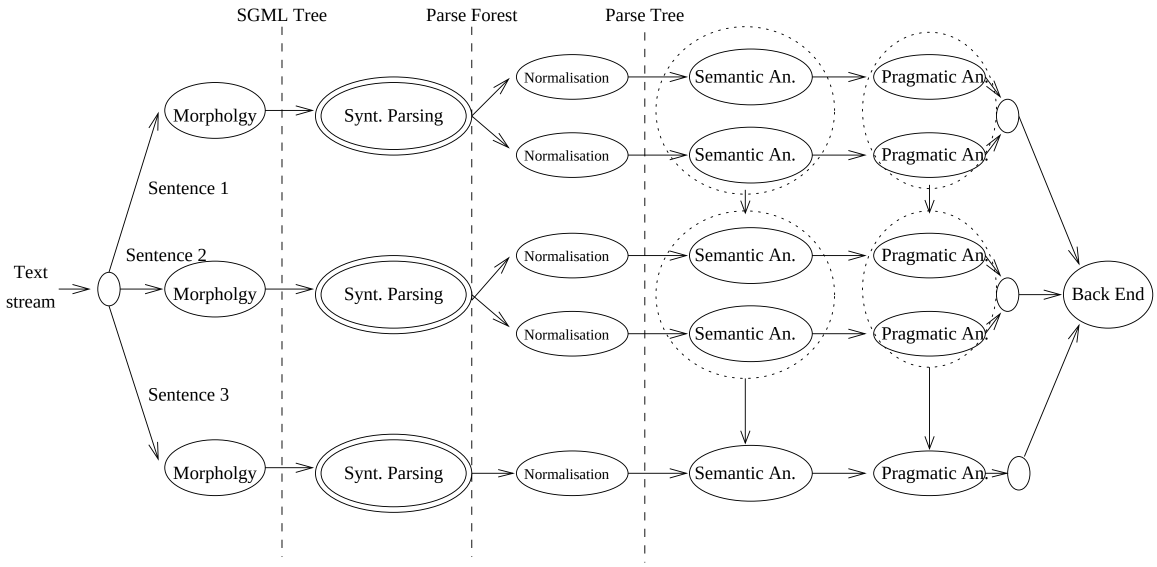


**Figure 4.14** Detailed structure of Lolita

possible to parallelise other sub-algorithms such as the graph algorithms operating on the semantic net. In fact, the most tedious part of the code changes was adding instances of `NFData` for intermediate data structures, which are spread over several dozen modules. However, in the meantime this process has been partially automated (Winstanley 1997).

### 4.5.7 Sun SPARCserver Implementation

This section discusses early performance measurements of Lolita on the Sun SPARC-Server. A realistic simulation showed an average parallelism between 2.5 and 3.1, using just the pipeline parallelism and parallel parsing. The actual speedup, however, does not exceed 2.4. Measurements with varying span values indicate that this is partly caused by fine-grained parallelism in the parsing stage. One obvious bottleneck in the computation is the sequential front end of about 10–15% caused by the C part of the syntactic parsing stage.

However, the wall-clock speedups obtained to date do not quite match the simulation results. As shown in Figure 4.15 a two processor execution on small inputs achieves an average parallelism of 1.4. A high span value is used to bound the amount of

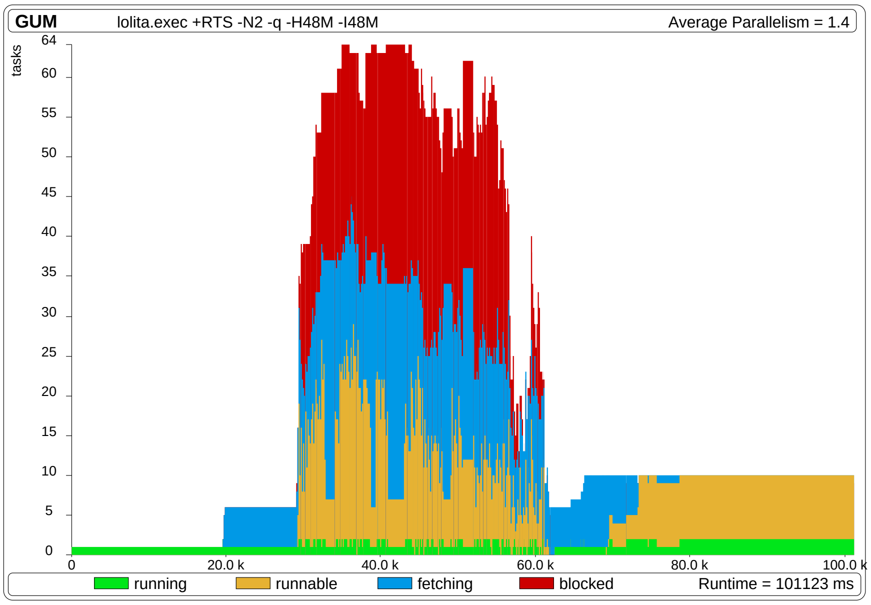


**Figure 4.15** Activity profile of Lolita run under GUM with 2 processors

parallelism in the parsing phase. This also bounds the total heap residency in the system, which proves to be very important. With more processors the available physical memory is insufficient and heavy swapping causes a drastic degradation in performance. The reason for this behaviour is that GUM, which is designed to support distributed-memory architectures uniformly, loads a copy of the entire code, and a separate local heap, onto each processor. Lolita is a very large program, incorporating large static data segments (totalling 16Mb), and requires 100Mb of virtual memory in total in its sequential incarnation.

One difference of the GUM activity profile in Figure 4.15 to the GranSim results is a larger degree of fetching in the former. This is probably caused by the rather expensive but generic communication routines used by PVM, on which GUM is based. In contrast, GranSim measures mainly the hardware costs for performing communication. Together with the fine granularity of the generated threads this increased overhead leads to a significantly smaller utilisation in the parsing stage. However, the later pipeline stages in the computation are still an effective source of parallelism.

## 4.6 LinSolv

The linear system solver discussed in this section uses an approach that is very common in the area of computer algebra: a *multiple homomorphic images* approach (Lauer 1982). This approach consists of the following three stages:

1. map the input data into several homomorphic images,
2. compute the solution in each of these images, and
3. combine the results of all images to a result in the original domain.

Since computer algebra algorithms aim at finding *exact solutions* to mathematical problems, unbounded data types like arbitrary precision integers are frequently used. In algorithms operating on arbitrary precision integers the original domain is typically $\mathbb{Z}$, the set of all integer values, and the homomorphic images are $\mathbb{Z}$ modulo $p$, written $\mathbb{Z}_p$, with $p$ being a prime number. The advantage of this approach becomes clear when the input numbers are very big and each prime number is small enough to fit into one machine word. In this case the basic arithmetic in the homomorphic images is ordinary fixed precision arithmetic with the results never exceeding one machine word. No additional cost for handling arbitrary precision integers has to be paid. Only in the combination phase will the big numbers appear again. In the case of $\mathbb{Z}$ as original domain the well-studied Chinese Remainder Algorithm (CRA) can be used in the combine step (Lipson 1971).

The linear system solver (LinSolv) discussed in this section uses such a multiple homomorphic images approach. Thus, it must be emphasised that this algorithm is *not meant to represent a highly-tuned numerical algorithm for finding just an* approximation *of a solution, but a typical symbolic algorithm for finding an* exact *solution*, which represents a wide class of computer algebra algorithms. Other algorithms with the same basic structure will be discussed in Section 4.7.

It is obvious that this approach lends itself to parallel processing: all solutions in the homomorphic images can be computed independently. An obvious bottleneck is the final combination stage. The following sections first discuss the structure of the sequential algorithm. Then a straightforward, parallel version is developed and improved by eliminating the two main sequential bottlenecks.

### 4.6.1 The Sequential Algorithm

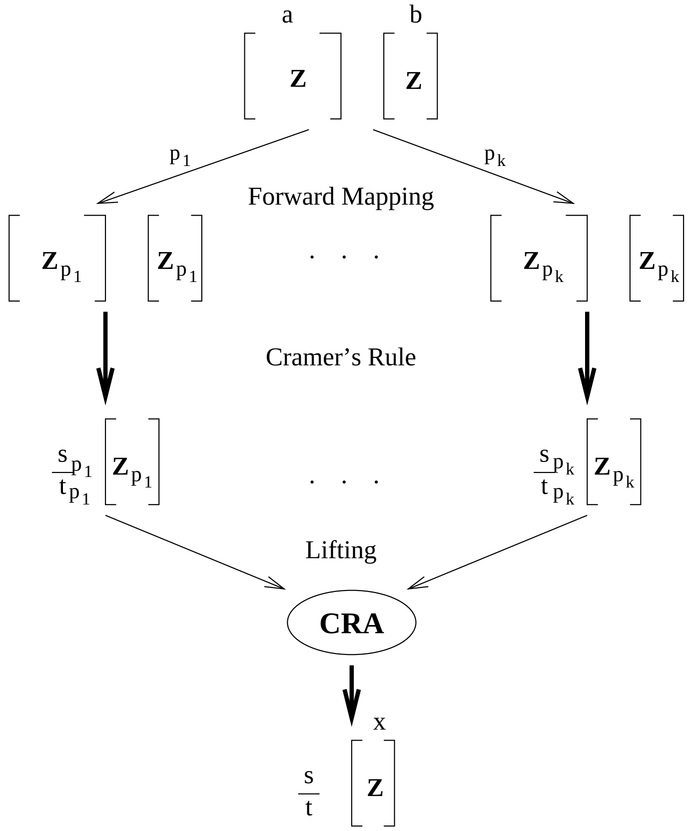


**Figure 4.16** Structure of the LinSolv algorithm

This section describes the basic structure of the sequential LinSolv algorithm. For a given matrix $a$ and vector $b$, both ranging over integers, this algorithm finds a solution $x$ to the equation $ax = b$. More formally, this problem can be specified as follows:

| | | | |
|---|---|---|---|
| **Input**: | $a, b$ | where | $a \in \mathbb{Z}^{n\times n},\ \det a \neq 0,\ b \in \mathbb{Z}^n$ |
| **Output**: | $s, t, x$ | where | $a(\frac{s}{t}x) = b,$ |
| | | | $s, t \in \mathbb{Z},\ x \in \mathbb{Z}^n$ |
| | | | $\gcd(s,t) = 1,\ \gcd_{i=1,\ldots,n} x_i = 1$ |

where $\mathbb{Z}$ denotes the set of all integers; for a domain $\mathbb{D}$ and an integer $n$, $\mathbb{D}^n$ denotes the set of all vectors of length $n$ with components from $\mathbb{D}$; and $\mathbb{D}^{n\times n}$ denotes the set of all 2-dimensional square matrices of size $n$ over $\mathbb{D}$. For an integer $n$, $\mathbb{Z}_n$ denotes the set of integers $\{0, \ldots, n-1\}$ (the homomorphic image of $\mathbb{Z}$ with base $n$). Note that we are computing a vector $x$ of integer values and factor out the rational part of the solution into $\frac{s}{t}$. This is convenient when using the result in a bigger application because later stages can avoid most of the expensive rational number arithmetic on the result vector.

A particularly important aspect of the algorithm we are designing is that it has to compute an exact solution over integers of arbitrary size. Therefore, the main questions to be considered for the efficiency of the sequential algorithm are:

1. How big are the intermediate values in the computation?
2. How high is the overhead associated with using rationals instead of integers?
3. Are there inherently sequential parts in the algorithm?

The first question is directly addressed by using a multiple homomorphic images approach, which bounds every value by the base of the image. The next two questions are crucial in picking a concrete algorithm for the solution phase. The following paragraphs discuss the individual stages of the algorithm with the paragraph on the solution phase discussing the advantages and disadvantages of three alternatives with respect to the questions raised above. Figure 4.16 summarises the overall structure of the algorithm.

**Forward mapping:** This stage is trivial: for a given prime number $p$ the function `` `mod` p`` is mapped over all elements of $a$ and $b$. This stage is easily parallelised.

**Homomorphic solutions:** We have investigated several candidates for computing the homomorphic solutions, which have the following characteristics:

- *Gaussian Elimination:* This is a very efficient algorithm often used for solving linear systems of equations. However, since it works over rational numbers the basic arithmetic operations are much more expensive than those over fixed precision integers. An alternative to the classical algorithm would be to introduce rational numbers only in the back-substitution phase by using for example Bareiss' variant of the algorithm. However, this variant requires $O(n^3)$ additional integer divisions, so it is not clear whether it gives an improved performance in practice.

- *LU-Decomposition:* The *LU-Decomposition* method has very strong data dependencies and yields an inherently sequential algorithm. An initial parallelisation of LU-Decompostion achieved only a speedup of 3.8 on an idealised machine. Some significant restructuring would be necessary to obtain an efficient parallel algorithm.

- *Cramer's Rule:* Although this algorithm is less efficient in the sequential case, it is very attractive because of its high potential of parallelism. In this algorithm the result is computed by evaluating $n+1$ independent determinants. The main structure of this algorithm is described below.

Iterative algorithms often used in numerical applications have not been considered because the goal here is to find an exact solution. Furthermore, LinSolv should use a parallel algorithm for computing a homomorphic solution in order to maintain scalability of the overall algorithm for cases where the number of available processors is higher than the number of homomorphic images used by the algorithm. Using an efficient sequential algorithm might achieve better results for small number of processors but is inherently limited in its parallelism.

The method used in LinSolv is based on *Cramer's rule.* This rule states that the solution of the equation $ax = b$ can be computed as a vector, with ratios of two determinants as components. In each component the denominator is the determinant of the original matrix $a$. The numerator of the $j$-th component is the determinant of the matrix obtained from $a$ by replacing the $j$-th column with the vector $b$. More formally, let $a_{p_i}$, $b_{p_i}$ be the homomorphic images of $a$ and $b$ w.r.t. the prime number $p_i$. Then the solution $x_{p_i} = [x_{p_{i1}}, \ldots, x_{p_{in}}]$ can be computed by:

$$x_{p_{ij}} = \frac{\det a'_{p_{ij}}}{\det a_{p_i}}$$

where $a'_{p_{ij}}$ is $a_{p_i}$ with the $j$-th column replaced with $b_{p_i}$.

When applying the above formula in a homomorphic domain $\mathbb{Z}_{p_i}$, the determinant $\det a_{p_i}$ might become 0. Obviously, no solution can be computed in such a domain. Prime numbers $p_i$ which result in $\det a_{p_i}$ being 0 are termed *unlucky* and must be filtered from the list of prime numbers which are used as bases for the homomorphic domains.

**Combination:** The final stage of the algorithm consist of combining the homomorphic solutions to a solution in the original domain $\mathbb{Z}$ ('lifting'). This combination can be done by using the Chinese Remainder Algorithm (CRA) (Lipson 1971). This algorithm finds the "original" of two images i.e. a value $r$, which maps to the given values $r_1, r_2$ in the images generated by the prime numbers $p_1, p_2$, respectively. More formally the algorithm can be specified as follows:

**Input**: $r_1, r_2, p_1, p_2$ where $p_1, p_2$ prime, $r_1 \in \mathbb{Z}_{p_1}$, $r_2 \in \mathbb{Z}_{p_2}$
**Output**: $r$ where $r \in \mathbb{Z}_{p_1 p_2}$, $r_1 = r \bmod p_1$, $r_2 = r \bmod p_2$

Although the CRA operation is associative, for two lists it is most efficient to use a left associative fold operation over the binary version above (Garner's algorithm (Knuth 1981, p.274)). The reason for this is that all computations in the binary CRA operate in the domain $\mathbb{Z}_{p_2}$, which can be chosen to be a fixed precision domain in each stage. Hence, the large accumulated input values $p_1$ and $r_1$ in the folding process are mapped to small numbers, making the binary CRA almost equally cheap in every step of the folding. Unfortunately, this is also an obvious sequential bottleneck.

Figure 4.17 shows the top level of the algorithm based on Cramer's rule. Note that `xList` is an infinite list of solutions in homomorphic images corresponding to prime numbers in the infinite list `primes`. The CRA computation itself is hidden in `list_cra`, which basically performs a left associative fold operation, accumulating the product of all prime numbers met so far until this product becomes larger than $s^n n!$ ($n$ is the size of the matrix $a$ and $s$ is the maximal element in $a$ and $b$). The `gen_xList` function has to check whether the modular determinant is 0 in order to avoid picking unlucky prime numbers. The strategy `strat` in the body of the let construct describes the dynamic behaviour of the code separately from the algorithmic code. For the sequential version the default strategy `rwhnf` can be used. Figures 4.19, 4.21, and 4.23, which are discussed in the subsequent sections, give different definitions of `strat` for parallel execution without changing the code in Figure 4.17 at all.

### 4.6.2 Naive Parallel Algorithm

Figure 4.18 shows a naive parallel version of LinSolv, written without strategies by parallelising `gen_xList`, which implements the forward mapping and solution phases. The idea of this code is to create a single parallel thread to evaluate both the forward

```
linSolv a b =
  let
    {- forward mapping and solution via Cramer's rule -}
    ...
    xList :: [[Integer]]  -- infinite list of solutions in hom images
    xList = gen_xList primes

    gen_xList (p:ps) =
       let
          modDet = toHom p (determinant (toHom p a))
          pmx = [ toHom p (determinant (replaceColumn j (toHom p a)
                                                         (toHom p b) ))
                | j <- [jLo..jHi] ]
          ((iLo,jLo),(iHi,jHi)) = bounds a
       in
       if modDet /= 0
         then (p : modDet : pmx)  : gen_xList ps
         else gen_xList ps

   {- combination via CRA -}
    ...
    detList = projection 1 xList

    det = list_cra pBound primes detList detList
    x_i i = list_cra pBound primes x_i_List detList
            where x_i_List = projection (i+2) xList

    x = map x_i [0..n-1]
  in
  x `using` strat
```

**Figure 4.17** Top level code of the sequential LinSolv algorithm

mapping (via `toHom`) and the determinant computations for each prime $p_i$. To achieve this behaviour a `parmap` function is used in the definition of a homomorphic solution `pmx`, and a `par` combinator is used in the body of the `let` construct to evaluate every homomorphic image in parallel. However, the actual dynamic behaviour is quite different: the thread sparked for `homsol` will only evaluate the top-level cons cell, which does not trigger the computation of the actual homomorphic solution (`pmx`) at all. Only when the result is required in the combination stage the `parmap` will be triggered, creating parallelism within a homomorphic image but sequentialising all stages. The combination stage is basically a `fold` operation. This causes a sequentialisation of the homomorphic images.

The resulting activity profile at the left hand side of Figure 4.20 reveals two stages

```
linSolv a b =
  let
    {- forward mapping and solution via Cramer's rule -}
    ...
    xList :: [[Integer]]  -- infinite list of solutions in hom images
    xList = gen_xList primes

    gen_xList (p:ps) =
     let
       ...
       homSol = (p : modDet : pmx)
       pmx = parmap ( \ j ->        -- parallelism within each hom im
                      let   a1 = replaceColumn j a0 b0
                      in    modHom p (determinant a1) )
                    [jLo..jHi]
       ((iLo,jLo),(iHi,jHi)) = matBounds a

       restList = gen_xList ps
     in
     if modDet == 0
       then gen_xList ps
       else par homSol (homSol : restList) -- par between hom ims

   {- combination via CRA -}
    ...
    detList = projection 1 xList

    det = list_cra pBound primes detList detList
    x_i i = list_cra pBound primes x_i_List detList
            where x_i_List = projection (i+2) xList

    x = map x_i [0..n-1]
  in
  x
```

**Figure 4.18** Naive parallel pre-strategy code

in the computation:

- In the first stage, up to approximately one third of the total execution time, the overall determinant det $a$ is computed using the same structure as for the overall computation. This causes a sequence of computations in the homomorphic domains, which is visualised as a sequence of small peaks.

- In the second stage, the solution is computed in each homomorphic image. All components of the solution are evaluated in parallel using a parallel determinant computation in each case. This yields a higher degree of parallelism within each

```
rnf det                                'seq'
seqListAccum 1 seq_sol_strat xList     'par'
parList rnf x
where
  seqListAccum :: Integer -> Strategy [Integer] -> Strategy [[Integer]]
  seqListAccum accum s =
    \ (xs:xss) -> if accum>pBound
                  then ()
                  else s xs 'seq'
                       seqListAccum (accum*(head xs)) s xss

  seq_sol_strat :: Strategy [Integer]
  seq_sol_strat = \ (p:modDet:pmx) -> rnf modDet 'seq'
                                      if modDet /= 0
                                        then seqList rnf pmx
                                        else ()
```

**Figure 4.19** Strategy version of a naive parallel LinSolv algorithm

stage.

Note that the number of parallel peaks in both stages is determined by the number of homomorphic images necessary to construct the result in the original domain (13 in this case).

The dynamic behaviour of this code becomes much clearer when reformulating the code with strategies. Figure 4.19 shows the definition of `strat` in the body of the `linSolv` function in Figure 4.17. Note that in contrast to the pre-strategy version the algorithmic code is unchanged. In the strategic version of the code it becomes clear that two nested strategies are used:

- the outer strategy, `seqListAccum` in this case, traverses the infinite list of solutions (`xList`), and
- the inner strategy, `seq_sol_strat` in this case, traverses the homomorphic solutions (`pmx`).

Each of these strategies can be done either sequentially or in parallel. From the above description of the dynamic behaviour of the naive parallel code it should be clear that both dimensions are done sequentially. The outer `seqListAccum` strategy encodes the dynamic behaviour of the algorithm when traversing `xList`: it accumulates the

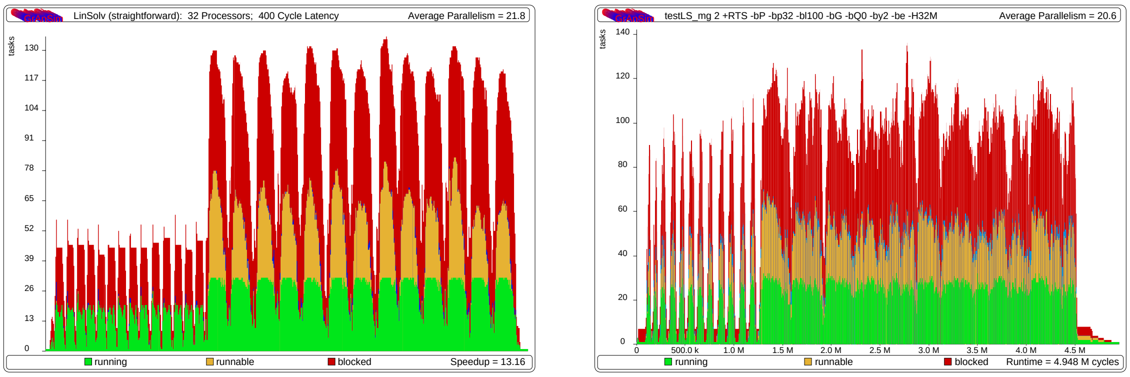


**Figure 4.20** Activity profile of pre-strategy and strategic naive LinSolv

product of all prime numbers in order to decide how many homomorphic solutions to generate. The explicit use of `seq` in `seqListAccum` reflects the evaluation order, which is implicit in the pre-strategy code. The inner `seq_sol_strat` strategy describes a dependency between the `modDet` component of the homomorphic solution and the rest. Although the `parmap` construct in Figure 4.18 specifies parallelism over the elements of the homomorphic solution, it is hidden by the first two elements of the result list in `sol`, which are demanded first when computing the overall determinant `det`. Figure 4.20 shows that the dynamic behaviours of the pre-strategy and the strategic version are almost identical.

### 4.6.3 Improved Version

Reflecting the performance tuning in the pre-strategy version of the code the strategy in Figure 4.21 shows two changes compared to the previous strategy: it does not force the computation of the determinant as a first step and it computes all components of the homomorphic solution in parallel using the `par_sol_strat` strategy. This avoids the delay in generating parallel processes for performing the most time consuming computations in the solution phase.

The activity profiles in Figure 4.22 show that the first stage of peaks has been merged with the second stage. The data dependency between the overall CRA and the homomorphic solutions has disappeared. However, by using the `seqListAccum` strategy over `xList` the combination stage is still sequential leading to regular drops in utili-

```
seqListAccum 1 par_sol_strat xList          `par`
parList rnf xs
where
  seqListAccum :: Integer -> Strategy [Integer] -> Strategy [[Integer]]
  seqListAccum accum s =
    \ (xs:xss) -> if accum>pBound
                    then ()
                    else s xs `seq`
                         seqListAccum (accum*(head xs)) s xss

  par_sol_strat :: Strategy [Integer]
  par_sol_strat = \ (p:modDet:pmx) -> rnf modDet `seq`
                                      if modDet /= 0
                                        then parList rnf pmx
                                        else ()
```

**Figure 4.21** Strategy version of an improved parallel LinSolv algorithm

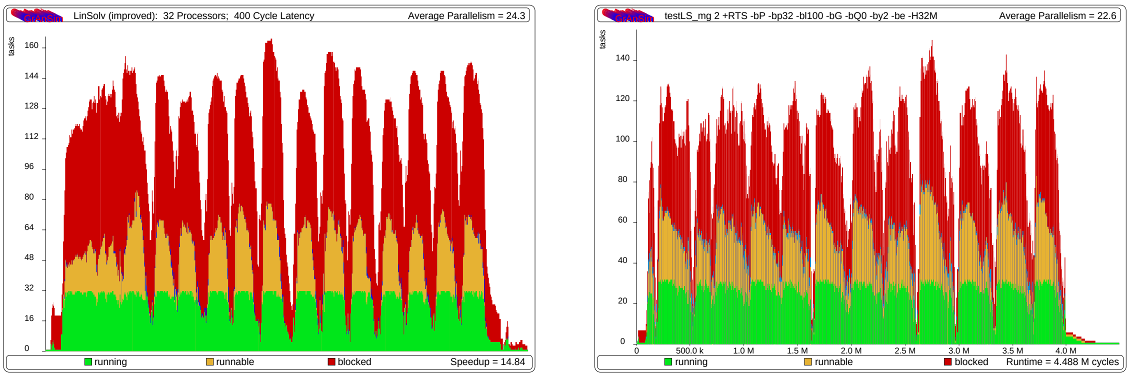


**Figure 4.22** Activity profiles of pre-strategy and strategic improved LinSolv

sation. In the pre-strategy code this corresponds to the dynamic behaviour generated by the `list_CRA` function.

### 4.6.4 Parallelism over the Homomorphic Images

The strategy in Figure 4.23 eliminates the sequential traversal of `xList` by guessing the number of primes needed to compute the overall result and using a `parListN` strategy to generate data parallelism over that segment of `xList`. Using `parList` inside the `par_sol_strat` strategy causes each component of the result to be evaluated

```
rnf noOfPrimes                                        `seq`
parListN noOfPrimes par_sol_strat xList               `par`
parList rnf xs
where
  par_sol_strat :: Strategy [Integer]
  par_sol_strat = \ (p:modDet:pmx) -> rnf modDet  `seq`
                                      if modDet /= 0
                                        then parList rnf pmx
                                        else ()
```

**Figure 4.23** Strategy of the final parallel LinSolv algorithm

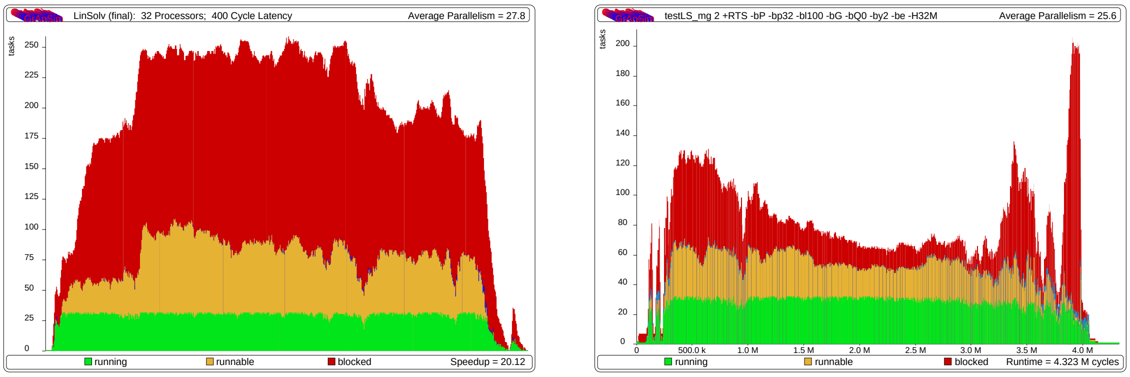


**Figure 4.24** Activity profiles of pre-strategy and strategic final LinSolv

in parallel. However, we still need the check for zero in order to avoid redundant computation. In order to minimise data dependencies in the algorithm we do not already check for unlucky prime numbers when computing `noOfPrimes`. If some prime numbers turn out to be unlucky the `list_cra` will evaluate more results by demanding a so far unevaluated list element. The final strategy application `parList rnf x` specifies that all elements of the result should be combined in parallel. Without this component there would be a sequence of combination steps at the end of the execution, one for each element in the result vector. In the activity profiles of Figure 4.24 the individual peaks have been merged into a period of consistently high utilisation.

This final version of LinSolv exhibits the highest average parallelism and lowest runtime of all strategic versions, reflecting the improved dynamic behaviour. Comparing the pre-strategy with the strategic versions in the activity profiles of Figures 4.20, 4.22, and 4.24, however, shows a slightly reduced average parallelism. This is due to small differences in the dynamic behaviour of both versions, in particular at the beginning and the end of the computation. More importantly, the main part of the computation shows the same dynamic behaviour in both versions. Based on previous measurements in assessing the overhead related to the use of evaluation strategies, it is unlikely that the lower average parallelism in the strategy version is due to this overhead.

### 4.6.5 Summary

Historically, the development and performance tuning of LinSolv predated the development of evaluation strategies. In hindsight the lack of separation between algorithmic and behavioural code severely complicated the program development. The most striking example is the `tree_CRA` algorithm we used in the pre-strategy version in order to guarantee parallelism between the homomorphic images. In order to handle an infinite list of solutions based on a guess how many solutions are needed, the `tree_CRA` algorithm keeps track of the number of unlucky primes and uses a "fail handler" in order to compute more results if necessary. This leads to the rather complicated algorithm in Figure 4.25, which combines the computation of the result with a specific dynamic behaviour suitable for parallelism. In contrast, the strategic version uses a much simpler sequential code, which is basically a fold operation which also tests for unlucky primes and accumulates the product of all lucky prime numbers. To add parallelism it is sufficient to change the `seqListAccum` in Figure 4.19 into a `parListN` in Figure 4.23. Again the different dynamic behaviour can be described by the top-level strategy.

It turns out that the additional parallelism of the combinations in `tree_CRA` does not improve the performance at all because combining two large values (in the nodes of the tree) is far less efficient than combining a large with a small value, which is done in each step of the `list_CRA`. Thus, although the `tree_CRA` generates parallelism at the end of the computation the total runtime actually increases. This can be seen in Table 4.2 where adding a tree CRA to the basic version of the algorithm, with a parallel determinant computation, does not further improve the efficiency of the algorithm. It only increases the total amount of work compared to a sequential version that uses a `list_CRA`. This behaviour of LinSolv corresponds to our experience with a parallel resultant algorithm using a similar multiple homomorphic images structure (see Section 4.7.2). However, this example shows that the use of strategies allows the

```
-- n ... guess on how many hom sols needed
-- ms ... infinite list of modules
-- as ... infinite list of values
-- ds ... infinite list of homomorphic determinants
tree_CRA :: Integer -> [Integer] -> [Integer] -> [Integer] ->
            (Integer, Integer)
tree_CRA n ms as ds =
 let
  res@(m, a, fails) = tree_CRA' ms' as' ds'
                      where ms' = take n ms
                            as' = take n as
                            ds' = take n ds
  handle_fails :: Integer -> Integer -> Integer ->
                  [Integer] -> [Integer] -> [Integer] -> (Integer, Integer)
  handle_fails n m a (m1:ms) (a1:as) (d1:ds)
   | n == 0    = (m, a)
   | d1 == 0   = handle_fails n m a ms as ds
   | otherwise = handle_fails (n-1) m' a' ms as ds
                 where
                   m'  = m * m1
                   a'  = par_binCRA m m1 inv a a1  -- NB: parallel version
                   inv = modInv m1 m
 in
  handle_fails fails m a ms as ds

-- here all lists are finite
tree_CRA' :: [Integer] -> [Integer] -> [Integer] ->
             (Integer, Integer, Integer)
tree_CRA' [p] [a] [0] = (1, 1, 1)  -- unlucky prime
tree_CRA' [p] [a] [_] = (p, a, 0)  -- normal case
tree_CRA' ps as ds =
  let
    n = length ps

    (left_ps, right_ps) = splitAt (n `div` 2) ps
    (left_as, right_as) = splitAt (n `div` 2) as
    (left_ds, right_ds) = splitAt (n `div` 2) ds

    left@(left_P, left_CRA, left_fails)  =
          tree_CRA' left_ps left_as left_ds

    right@(right_P, right_CRA, right_fails)  =
          tree_CRA' right_ps right_as right_ds

    inv = modInv right_P left_P
    cra = par_binCRA left_P right_P
                     inv left_CRA right_CRA
  in
    left  `par` right `par` inv `par` (
    cra   `seq`          -- force computation of cra first
            (left_P * right_P,
             cra,
             left_fails + right_fails) )
```

**Figure 4.25** A tree CRA used in the pre-strategy version programmer to explore different variants of the parallel code without performing a major restructuring of the algorithm.

Table 4.2 compares the runtimes (in kilocycles), average parallelism, total amount of work (as percentage compared to the work in the sequential setup), and speedups for the three versions discussed above with different setup variants. Although this table only records the results from the strategic versions, it reflects the pre-strategy versions as well, because they show corresponding runtime behaviour as demonstrated in Figures 4.20, 4.22 and 4.24. Overall the three stages of the parallelisation, from a naive to the final version, show an increasing average parallelism and speedup. The percentage of total work is roughly unchanged, indicating that no speculative parallelism is added during the performance tuning. In each of the three stages the best results are obtained from a setup using a parallel determinant computation in the solution stage. However, the parallel determinant computation performs some redundant work shown by the constantly high percentage of total work. This is mainly due to repeated traversals of data structures when constructing the sub-matrices defined in Cramer's Rule.

**Table 4.2** Measurements of all versions of LinSolv

| Setup | Runtime (kilocycles) | Average Parallelism | Total Work | Speedup |
|---|---|---|---|---|
| | *Naive parallel algorithm* | | | |
| Sequential | 78,651 | | | |
| Par Determinant (default) | 4,948 | 20.6 | 130% | 15.9 |
| Par Determinant & Tree CRA | 5,509 | 20.6 | 144% | 14.3 |
| | *Improved algorithm* | | | |
| Sequential | 78,651 | | | |
| Par Determinant (default) | 4,488 | 22.6 | 129% | 17.5 |
| Par Determinant & Tree CRA | 5,675 | 20.0 | 144% | 13.9 |
| | *Parallelism over homomorphic images* | | | |
| Sequential | 78,651 | | | |
| Par Determinant (default) | 4,323 | 25.6 | 141% | 18.2 |
| Par Determinant & Tree CRA | 5,130 | 22.1 | 144% | 15.3 |

As further work it would be interesting to compare this LinSolv version with one using a Gaussian elimination algorithm in the solution phase. Such an implementation would use rational arithmetic rather than integer arithmetic. The tighter data dependencies would probably reduce the parallelism inside the solution stage. How-

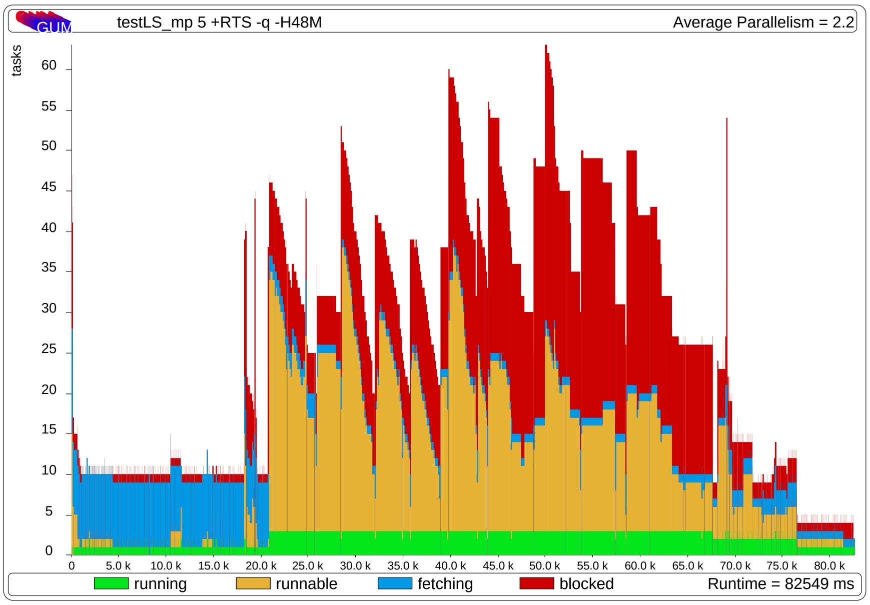


**Figure 4.26** Activity profile of LinSolv in a 3 processor GUM setup

ever, the overall structure of the parallelism generated by the multiple homomorphic images approach should be unchanged. Therefore, the final strategy developed in this section can be re-used.

Additionally to the these measurements under GranSim, the final version of LinSolv has been run under GUM on a 4 processor SUN shared-memory machine. Because of competing processes on that machine, only up to 3 processors have been used in the timings. As a result we obtained relative speedups, i.e. speedups of the parallel execution compared to a 1 processor GUM execution, of 1.67 on 2 processors and 2.10 on 3 processors. For this program the single processor efficiency is 78%, i.e. the optimised sequential version finished within 78% of the runtime for the 1 processor GUM version. This matches with previous experiences that report an efficiency of around 80% for most GUM programs. The absolute speedups for LinSolv, i.e. the speedups of the parallel execution compared to an optimised single processor execution, are 1.30 on 2 processors and 1.66 on 3 processors. Figure 4.26 shows an activity profile of running LinSolv in a 3 processor setup on the shared-memory machine.

## 4.7 Comparison with Parallel Imperative Programming

This section gives a comparison of the programming style in parallel imperative programming with a style of strategic parallelism as elaborated in this chapter. All of the algorithms in this section are computer algebra algorithms implemented in the Paclib system. This system combines a kernel for handling light-weight threads with a runtime system for garbage collected memory management and a library for basic computer algebra operations (Hong et al. 1992), all written in C. It has been implemented on a Sequent Symmetry shared memory system based on Intel i386 processors. All of the measurements have been performed on 16 processors.

### 4.7.1 LinSolv

Before attempting a functional solution to LinSolv, the author has previously implemented both sequential and parallel imperative solutions in C (Loidl 1993). The parallel version required significant restructuring, in order to eliminate sequential control dependencies. As an example, Figure 4.27 shows the code for managing the parallelism in the forward mapping and solution stages of LinSolv. The primitives `pacStart` and `pacWaitListRm` are used for starting and synchronising threads, respectively. The function `Solve` computes a homomorphic solution. Note the destructive use of the list-processing functions `COMP` (cons), `ADV` (tail) etc, to create a list of tasks, which must be explicitly manipulated by the programmer. Results are extracted non-deterministically from this list and combined in later stages of the algorithm.

Using GpH and GranSim, the code in Figure 4.27 could be written much more simply as:

```
(xList, pList) = unzip (parMap rnf (solve a b detA n) primeList)
```

The strategic code avoids explicitly specifying when to create threads and when to synchronise them. These decisions are made by the runtime-system. Of course, this straightforward translation of the imperative code does not enforce the sophisticated order of evaluation produced by the strategy in Figure 4.23. However, the same order

```
Step2:
 /* Forward mapping and solution in homomorphic images */

 taskList = NIL;
 while (!ISNIL(primeList)) {
   /* Extract the next prime from primeList */
   ADV(primeList,&p,&primeList);

   /* Create a task to solve each p in parallel */
   t = pacStart(Solve,5,A,B,detA,n,p);
   taskList = COMP(t,taskList);
 }

 /* Collect the results */
 X = NIL; pList = NIL; xList = NIL;
 while (!ISNIL(taskList)) {
   /* Wait for the first task to complete  */
   r = pacWaitListRm(&taskList);

   /* Deconstruct the result tuple */
   p = FIRST(r); X = SECOND(r);

   /* xList is the list of result vectors */
   xList = COMP( X, xList );

   /* pList is the list of primes which were used */
   pList = COMP( p, pList );
 }
```

**Figure 4.27** PACLIB code of generating and synchronising processes in LinSolv

of evaluation has to be coded into the imperative algorithm, in the function `Solve`, too.

Important differences in the parallel structure of the Haskell and the C versions of the code are caused by the different semantics of both languages and by the level of detail that has to be specified for describing a parallel algorithm. The C version required more restructuring in order to avoid synchronisation barriers between the stages of the algorithm. In the C version several variants of the parallel CRA have been implemented. In particular, these changes were much simpler in the functional code. This observation suggests to use strategies and a functional language to prototype

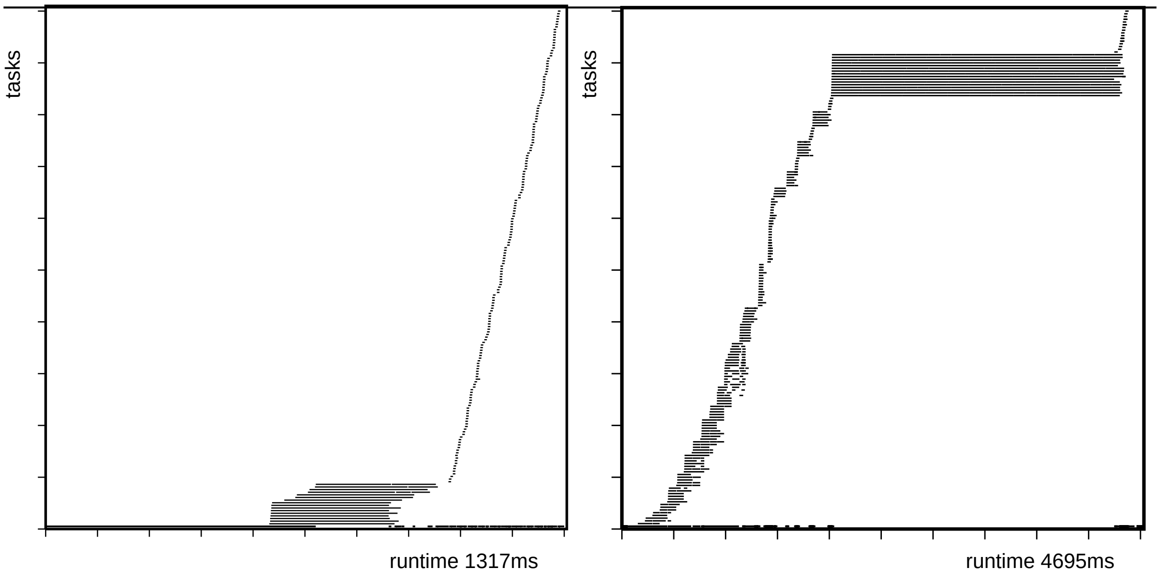


**Figure 4.28** Per-thread activity profiles for imperative LinSolv and parallel p-adic computation

parallel algorithms, which might then the translated back into an imperative language if necessary. We have taken this approach of parallel prototyping for example in Hall et al. (1997).

Figure 4.28 shows, on the left hand side, the per-thread activity profile for the imperative version of LinSolv on a 16-processor Sequent Symmetry. These should not be seen as direct comparisons with the graphs in Section 4.6 since they are based on a naive implementation. Furthermore, the C version has a much coarser, hand-tuned, granularity than the functional code discussed in Section 4.6. However, it is interesting to observe the barrier between the bulk of the parallel computation and the fine-grained back-end of the computation. In contrast, the Haskell version achieves some pipeline parallelism between these stages for free, i.e. without restructuring of the code.

### 4.7.2 Parallel Resultant Computation

In Hong & Loidl (1994) the author has contributed to the implementation and measurement of five versions of a parallel resultant algorithm. A resultant of two $r$ variate polynomials is the determinant of a special matrix constructed out of the coefficients of these polynomials, a so-called "Sylvester matrix". The entries in the matrix are $r-1$ variate polynomials and so will be the overall determinant.

The algorithm itself has a multiple homomorphic images structure, but in contrast to LinSolv it works over multivariate polynomials. In this case, each $r$-variate polynomial is mapped into an $r-1$ variate polynomial by evaluating the main variable at a given point, which acts as the basis for the homomorphic image. In the combination phase, an interpolation algorithm with a structure similar to the CRA algorithm has to be used.

The different variants of the parallel resultant algorithm show typical characteristics of algorithms with a multiple homomorphic images structure:

1. In Variant 1 a tree-based interpolation gives poor results, compared to a list-based version, because of the additional complexity of this operation.
2. In Variant 2 a different computation structure has been used, involving a very time consuming matrix inversion. In this version a global synchronisation on the strict data structure is necessary before the list-structured interpolation can commence. This causes a sequential barrier in the evaluation.
3. In Variant 3 an explicit threshold is used in the parallel list-structured interpolation algorithm in order to avoid the generation of too fine-grained threads in the combination stage.

As a result of our performance measurements the rather fine-grained Variant 3, with an experimentally tuned threshold value proved to be the most efficient version.

### 4.7.3 Parallel P-Adic Computation on Rational Numbers

The goal of p-adic computation is to speed-up basic arithmetic on e.g. rational numbers by using an alternate representation of these numbers, namely a "Hensel code", and by defining the basic arithmetic over Hensel codes. A Hensel code is a truncated power series with a prime number $p$ as base and a fixed length $r$. The Extended Euclidean Algorithm (EEA) can be used for the forward mapping stage. A p-adic computation then uses the multiple homomorphic images approach by choosing several Hensel codes with varying prime numbers $p$, using the redefined basic arithmetic in each image to compute a solution, and by combining Hensel codes into a rational number again by using the CRA and a translation algorithm from Hensel codes into rational numbers.

In joint work the author has implemented translations of rational numbers to and from Hensel codes, and basic arithmetic operations over Hensel codes. In Limongelli & Loidl (1993) we have measured the efficiency of basic operations over rational numbers using this p-adic approach. Two versions of the combination step have been tested: SCA, which applies the CRA to every digit of the resulting Hensel codes, yielding a Hensel code representing the result which is then translated into a rational number; and PCA, which first translates the Hensel codes into rational numbers and then applies the CRA to these rational numbers. Note, that the structure of SCA is the same as the combination stage of LinSolv in Figure 4.17, where a list-structured CRA is applied to projections onto the list of result vectors.

The measurements with these algorithms have shown that PCA, which only requires global synchronisation once at the end of the CRA is more efficient than SCA, which requires a global synchronisation for every digit of the Hensel code. Again, this is in part due to the strict data structures used in the computation, which prohibit a straightforward pipelining of these stages. The right hand side of Figure 4.28 shows a per-thread activity profile of PCA.

## 4.8 A Methodology for Parallel Non-Strict Functional Programming

Based on the experiences in parallelising the programs discussed in this chapter and more programs discussed in Trinder et al. (1998) and Hall et al. (1997), an emerging methodology for parallelising large non-strict functional programs is outlined below. In the meantime, this methodology has also been used by other researchers for example in the parallelisation of the parallelising compiler Naira (Junaidu 1998). The approach is top-down, starting with the top-level pipeline, and then parallelising successive components of the program. The first five stages are machine-independent. This approach uses several ancillary tools, including time profiling (Sansom & Peyton Jones 1995) and the GranSim simulator (Hammond et al. 1995). Several stages use GranSim, which is fully integrated with the GUM parallel runtime system (Trinder, Hammond, Mattson Jr., Partridge & Peyton Jones 1996). A crucial property of GranSim is that it can be parameterised to simulate both real architectures and an idealised machine with, for example, zero-cost communication and an infinite number of processors.

The stages in this methodology, whose overall structure is similar to others used for large-scale parallel functional programming (Hartel et al. 1995), are as follows.

1. **Sequential implementation.** Start with a correct implementation of an inherently-parallel algorithm or algorithms.

2. **Parallelise Top-Level Pipeline.** Most non-trivial programs have a number of stages, e.g. lex, parse and typecheck in a compiler. Pipelining the output of each stage into the next is very easy to specify, and often gains some parallelism for minimal change.

3. **Time Profile** the sequential application to discover the "big eaters", i.e. the computationally intensive pipeline stages.

4. **Parallelise Big Eaters** using evaluation strategies. It is sometimes possible to introduce adequate parallelism without changing the algorithm; otherwise the algorithm may need to be revised to introduce an appropriate form of parallelism, e.g. divide-and-conquer or data-parallelism.

5. **Idealised Simulation.** Simulate the parallel execution of the program on an idealised execution model, i.e. with an infinite number of processors, no communication latency, no thread-creation costs etc. This is a "proving" step: if the program is not parallel on an idealised machine it will not be on a real machine. We now use GranSim, but have previously used HBCPP. A simulator is often easier to use, more heavily instrumented, and can be run in a more convenient environment, e.g. a workstation.

6. **Realistic Simulation.** GranSim can be parameterised to closely resemble the GUM runtime system for a particular machine, forming a bridge between the idealised and real machines. A major concern at this stage is to improve thread granularity so as to offset communication and thread-creation costs.

7. **Real Machine.** The GUM runtime system supports some of the GranSim performance visualisation tools. This seamless integration helps understand real parallel performance.

## 4.9 Related Work

### 4.9.1 Evaluation Strategies

This section discusses the relationship of evaluation strategies to similar programming techniques proposed in the literature.

#### Algorithmic Skeletons

A skeleton (Cole 1989) is a higher-order function that is parameterised with sequential sub-programs and that specifies a certain commonly encountered parallel structure. The most commonly encountered skeletons are pipelines and variants of the common list-processing functions such as `map`, `scan` and `fold`. A general treatment has been provided by Rabhi, who has related algorithmic skeletons to a number of parallel paradigms (Rabhi 1995).

Since a skeleton is simply a parallel higher-order function, it is straightforward to write skeletons using strategies. For example the `parMap` function in Section 4.3.4 is a skeleton. A more elaborate divide-and-conquer skeleton, based on a Concurrent Clean function (Nöcker, Smetsers, van Eekelen & Plasmeijer 1991) can be written as follows. It should be noted that all of these strategic skeletons are much higher-level than the skeletons used in practice, which have a careful implementation giving good data distribution, communication and synchronisation. As mentioned before, the aspect of data distribution is currently not directly controlled by strategies. The explicit function application operator `$`, although not absolutely necessary, is used to make the application of a strategy explicit in the code.

```
divConq :: (a -> b) -> a -> (a -> Bool) ->
           (b -> b -> b) -> (a -> Bool) -> (a -> (a,a)) -> b
divConq f arg threshold conquer divisible divide
   | not (divisible arg) = f arg
   | otherwise     = conquer left right `demanding` strategy
    where
      (lt,rt)  = divide arg
      left     = divConq f lt threshold conquer divisible divide
      right    = divConq f rt threshold conquer divisible divide
      strategy = if threshold arg
                   then (seqPair rwhnf rwhnf) $ (left,right)
                   else (parPair rwhnf rwhnf) $ (left,right)
```

Many strategic functions take the opposite approach to skeletons: a skeleton parameterises the control function over the algorithm, i.e., it takes sequential sub-programs as arguments. However, a strategic function may instead specify the algorithm and parameterise the control information, i.e. take a strategy as a parameter.

It is also possible to combine skeletons with imperative approaches. For example, the Skil (Botorog & Kuchen 1996) compiler integrates algorithmic skeletons into a subset of C (C-). The performance of the resulting program is close to that of a hand-crafted C- application.

### Coordination Languages

Coordination languages build parallel programs from two components: the *computation* model and the *coordination* model (Gelernter & Carriero 1992). Like evaluation strategies, programs have both an algorithmic and a behavioural aspect. It is not necessary for the two computation models to be the same paradigm, and in fact the computation model is often imperative, while the coordination language may be more declarative in nature. It is sometimes useful to distinguish two kinds of coordination languages. *Embedded coordination languages*, such as Linda, perform coordination via calling certain coordination primitives from within the computational code. In contrast, *embedding coordination languages* specify a parallel framework of the program execution with sequential sub-algorithms. As the development of the algorithms in this chapter shows, strategies can be used in both styles but they suggest a top-down parallelisation corresponding to the use of an embedding coordination language. The original model of directly using `seq` and `par` in GPH is, in contrast, closer to an embedded language, with constructs for parallelism scattered throughout the code.

PCN (Foster & Taylor 1994) composes tasks by connecting pairs of communication ports, using three primitive composition operators: sequential composition, parallel composition and choice composition. It is possible to construct more sophisticated parallel structures such as divide-and-conquer, and these can be combined into libraries of reusable templates. This approach is much more explicit than evaluation strategies, and, similarly to the other systems described here, it is possible to introduce deadlock.

Linda (Gelernter & Carriero 1992) is built on a logically shared-memory structure. Objects (or *tuples*) are held in a shared area: the Linda *tuple space*. Linda processes manipulate these objects, passing values to the sequential computation language. In the most common Linda binding, C-Linda, this is C. Sequential evaluation is therefore performed using normal C functions.

Darlington et al. (1995) integrate the coordination language approach with the skeleton approach, providing a system for composing skeletons, SCL. SCL is basically a data-parallel language, with distributed arrays used to capture not only the initial data distribution, but also subsequent dynamic redistributions. SCL introduces three kinds of skeleton: configuration, elementary and computational skeletons. Configuration skeletons specify data distribution characteristics, elementary skeletons capture the basic data parallel operations as the familiar higher-order functions `map`, `fold`, `scan` etc. Finally, computational skeletons add control parallel structures such as farms, SPMD and iteration. It is possible to write higher-order operations to transform configurations as well as manipulate computational structures etc.

Based on the same concept, $P^3L$ (Pelagatti 1993) defines a set of parallel constructs, each of which abstracts a specific form of commonly used parallelism. $P^3L$ integrates the concept of skeletons and the PCN model. The latter is used for describing details of the parallel execution of the skeletons.

### Parallel Language Extensions

Rather than providing completely separate languages for coordination and computation, several researchers have instead extended a functional language with a small, but distinct, process control language. This can be simply a set of annotations as it is used by Burton (1984), in $Hope^+$ (Kewley & Glynn 1989) and in Concurrent Clean (Nöcker, Smetsers, van Eekelen & Plasmeijer 1991). Most closely related to strategies, and therefore discussed in more detail here, are Caliban (Kelly 1989) and first-class schedules (Mirani & Hudak 1995).

**Caliban.** The Caliban system developed by Kelly (1989) bears a strong resemblance to evaluation strategies in its separation of algorithm and parallelism. Corresponding to the `using` construct in strategies, Caliban introduces a `moreover` construct to describe the parallel control component of a program. Frequently higher-order functions are used to structure the process network, corresponding to higher-order strategies such as `parList`.

One fundamental difference to strategies is that constructs in the `moreover` clause represent a separate language to the computation language. In particular, all values in such a clause must be resolved at compile time, thus representing a static description of the parallel structure. The values in a `moreover` clause are explicit process names. In a strategy, however, variable names, representing thunks in the program execution, can be used to avoid introducing additional names that are not necessary for understanding the structure of the program. Similarly to PCN, Caliban gives explicit description of the connections between the processes. Thereby, it can construct complex networks of processes but it may also introduce deadlock.

For example, the following function defines a pipeline. The □ syntax is used to create an anonymous process which simply applies the function it labels to some argument. The `arc` constructs indicates a wiring connection between two processes. The `chain` construct creates a chain of wiring connections between elements of a list. The result of the pipeline function for a concrete list of functions and some argument is thus the composition of all the functions in turn to the initial value. Moreover, each function application is created as a separate process.

```
pipeline fs x = result
  where    result = (foldr (.) id fs) x
  moreover (chain arc (map (□) fs))
           /\ (arc □(last fs) x)
           /\ (arc □(head fs) result)
```

**Para-Functional Programming.** Para-functional programming (Hudak 1986) is an extension to the functional programming paradigm that allows to express operational details like scheduling or mapping by annotating program expressions with constructs of a separate process control language. The latter specifies the scheduling and the mapping of parallel processes. One important advantage of this approach is that it can be used with any functional language. The following description uses Hudak's syntax for para-functional programming in Haskell (Hudak 1991).

With the annotations provided by a para-functional programming system it is possible to specify

- an *evaluation order* of the program and
- a *mapping* of a program to a machine.

*Controlling Evaluation Order.* The default evaluation order is lazy evaluation. However, this can be changed for any expression in the program by using a *scheduled expression* of the following form:

*exp* `sched` *sched-exp*

where *exp* is a program expression and *sched-exp* is a schedule. Note that a subexpression in *exp* can be labelled by using a labelled expression of the form *lab*`@`*exp*.

A schedule defines the evaluation order and the parallelism obtained when evaluating the expression. To this end, three kinds of primitive schedules are defined for a labelled expression:

- The *demand* for the evaluation of *exp*, denoted by `D`*lab*,
- the *start* of the evaluation of *exp*, denoted by `^`*lab*,
- the *end* of the evaluation of *exp*, denoted by *lab*`^`.

Note that a value can be demanded several times, but it can only be evaluated once. The following operations can be used to combine schedules:

- `s1.s2` denotes the *concatenation* of the schedules `s1` and `s2` (sequential composition);
- `s1|s2` denotes the *concurrence* of the schedules `s1` and `s2` (parallel composition).

The following examples of scheduled expressions describe their operational behaviour in more detail:

- `(e0 m@e1 n@e2) sched Dm|Dn`. This expression specifies a parallel demand for the evaluations of the expressions `e1` and `e2`. Because it is not guaranteed that these values will be needed in the evaluation of `e0`, this schedule denotes speculative parallelism.

- `o@(l@e0 m@e1 n@e2) sched l.m.n.o`. This expression specifies a left-to-right call-by-value semantics. Note that in this expression the schedule `lab` is the abbreviation for `Dlab.lab^`. However, this schedule does not prohibit parallelism inside `e0`, `e1` or `e2`.

*Mapping an Expression to a Machine.* In order to specify a mapping of the evaluation of expressions to processors *mapped expressions* of the following form is used:

*exp* `on` *pid*

where *exp* is a program expression and *pid* is the identifier of the processor on which the expression will be evaluated.

Such an expression can be used for example to evaluate the two components of an addition on two different processors:

```
(f x on 0) + (g y on 1)
```

With this expression the function call `f x` will be evaluated on processor `0` and the function call `g y` will be evaluated on processor `1`. Note, that since `+` is a strict operation, both function calls will be evaluated in parallel due to the default evaluation strategy of lazy evaluation.

It is also possible to use functions in computing the processor identifier. Thereby, a mapping that is relative to the current processor can be realised. For that purpose the predefined identifier `self` always contains the identifier of the current processor.

**First-Class Schedules.** First-Class schedules (Mirani & Hudak 1995) combine para-functional programming with a monadic approach. Where para-functional schedules and mapped expressions are separate language constructs, first-class schedules are fully integrated into Haskell. This integration allows schedules to be manipulated as normal Haskell monadic values.

The primitive schedule constructs and combining forms are similar to those provided by para-functional programming. The schedule `d` *exp* demands the value of expression *exp*, returning immediately, while `r` *exp* suspends the current schedule until *exp* has been evaluated. Both these constructs have type `a -> OS Sched`. Similarly, both the sequential and parallel composition operations have type `OS Sched -> OS Sched -> OS Sched`. The monadic type `OS` is used to indicate that schedules may interact in a side-effecting way with the operating system.

Rather than using a language construct to attach schedules to expressions, Mirani and Hudak instead provide a function `sched`, whose type is `sched :: a -> OS Sched -> a`, and which is equivalent to the `using` function in evaluation strategies. The `sched` function takes an expression *exp* and a schedule *sched*, and executes the schedule. If the schedule terminates, then the value of *exp* is returned, otherwise the value of the *sched* application is $\perp$. There are also constructs to deal with task placement and dynamic load information which have no equivalent strategic formulation.

In evaluation strategy terms, both the `d` and `r` schedules can be replaced by calls to `rwhnf` without affecting the semantics of those para-functional programs that terminate. Unlike evaluation strategies, however, with first-class schedules it is also possible to suspend on a value without ever evaluating it. Thus, para-functional schedules can give rise to deadlock in situations which cannot be expressed with evaluation strategies. A trivial example might be:

```
f x y = (x,y) `sched` r x . d y | r y . d x
```

Compared with evaluation strategies, it is not possible to take as much direct advantage of the type system: all schedules have type `OS Sched` rather than being parameterised on the type of the value(s) they are scheduling.

There can also be a loss of referential transparency when using schedules, since expressions involving *sched* may sometimes evaluate to $\perp$, and other times to a non-$\perp$

value. This can happen both through careless use of demand and wait, as in the deadlock-inducing example above, and conceivably if dynamic load information is used to demand an otherwise unneeded value. If the program terminates (yields a non-$\perp$ value), however, it will always yield the same value.

### 4.9.2 Large-Scale Parallel Functional Programming

#### Non-Strict Languages

Previous experience with parallelising large non-strict functional programs using an annotation based approach has shown that efficient parallel execution without explicit control of parallelism is possible. In particular the FLARE project (Runciman & Wakeling 1995) studied several large parallel applications. For example the parallelisation of a computational fluid dynamics simulation (Grant et al. 1995) demonstrated the ease of parallelisation compared to an imperative version of the program. The necessary changes were localised in a few functions. However, these functions did not appear in top-level modules, but were part of crucial sub-modules. Therefore, a deeper understanding of the code and its dynamic behaviour was necessary. This case study also emphasised the importance of a sophisticated parallel engineering environment. In the meantime the development of GranSim, GUM, and a set of visualisation tools has significantly improved this environment. Corresponding to other experiences with parallelising large programs in non-strict languages the heap consumption turned out to be one of the biggest problems for the efficiency of the program.

The toolkit for parallel functional programming discussed in Hartel et al. (1995) is very similar to our parallel programming methodology (see Section 4.8). It uses both an interpreter and a compiler for sequential debugging. A simulator supports parallel simulation in three levels of detail. A compiler produces platform independent parallel code. However, our system differs in the following aspects. The compilation of GpH programs is performed by GHC, a state-of-the-art optimising compiler rather than a prototype compiler with limited support for code optimisation. Furthermore, GHC provides the innovative cost-centre technology of profiling sequential lazy code, which has proven to be essential to understand the performance of the sequential program. Rather than using an annotation based approach, strategies support a top-down parallelisation of the code and since strategies are Haskell functions they can use the full power of the language, such as higher-order functions and polymorphism. Finally, the use of a sandwich annotation in Hartel et al. (1995), which fully evaluates two arguments in parallel before they are combined, favours divide-and-conquer parallelism. Pipeline parallelism, which naturally arises in a lazy language, has to be transformed via a set of semi-automatic transformations. As an example of parallelising a large program, Hartel et al. (1995) discuss a tidal prediction program. This program is an application from the area of numerical scientific computation. In the parallelisation of the program a new "communication lifting" transformation is used in order to exploit wavefront parallelism in a grid performing computational fluid dynamics operations (solving partial differential equations). Thus, the overall parallel structure is a pipeline of iteration steps with massive data parallelism within each step.

Shaw et al. (1996) discuss the performance tuning of a global ocean circulation model implemented in Id. In contrast to the previously discussed languages, Id uses parallel eager computation to exploit parallelism. In practice this approach exposes more parallelism and reduces the heap consumption of the program. However, it often creates speculative computation, which might waste a significant amount of resources. This program, which has originally been written in FORTRAN and executed on a CM-5, has a regular control structure but an irregular data structure. This is in contrast to our applications, which come from the symbolic computation area and typically have a less regular control structure. The performance tuning process of this algorithm uses explicit compiler pragmas to force loop unrolling. In order to modify the granularity of the generated parallelism k-bounded loops are used. However, with this construct it is necessary to consider all k-bounds in the program in order to obtain a good parallel behaviour. Clearly, this behaviour poses problems for modular parallel program development.

Sur & Böhm (1994*a*) show that the non-strict semantics of Id allows a very natural formulation of producer-consumer parallelism in two central stages of the Dongarra-Sorensen Eigensolver. In previous papers, this kind of parallelism has been reported difficult to achieve for imperative languages. This reflects our observation that non-strict languages suggest the use of pipeline parallelism, because of the lack of a barrier synchronisation between the pipeline stages.

Several case studies for parallelising non-strict functional programs reported problems with excessive heap consumption. Sometimes running the parallel program on the full input was not possible (Blelloch & Narlikar 1997). In several cases impure features have been used to reduce the heap consumption e.g. Sur & Böhm (1994*b*) and Hammes et al. (1995). We have observed similar problems of resource consumption, in particular heap consumption, in the parallelisation of Lolita (see Section 4.5).

### Strict Languages

Michaelson & Scaife (1995) describe the implementation of several components in a parallel vision system to recognise 3D objects in a 2D scene from intensity data. The parallel algorithms, which are finally executed as Occam2 programs on a Meiko multi-processor, are prototyped in SML. Special emphasis is put on combining several components into a large-scale system and analysing the resulting performance out of this combination. The parallelisation uses skeletons in particular a farm skeleton to realise a parallel map. The main data structure in this case is a nested list, and data-oriented parallelism is used. In the SML prototype some form of pre-loading data onto a processor is achieved by using partial applications consisting of the function to be computed and the data to be pre-loaded. No explicit locality information has to be added. An interesting observation made in Michaelson & Scaife (1995) is that if computation dominates communication the load balance becomes more important. This directly corresponds to our experience with rescheduling schemes discussed in Section 3.3.1: for low-latency systems, where the communication is rather cheap and the computation comparatively expensive, the load balance is more important than the data locality in the system. The overall parallel structure of the parallel vision system is a pipeline with processor farms, representing data parallelism, in each component.

Skeleton-based approaches (Darlington et al. 1995) often suffer from problems of compositionality similar to the k-bounded loops approach discussed above: it is hard to construct an efficient parallel program out of efficient parallel components. The root of the problem is that although individual skeletons represent optimised parallel code, the composition of several skeletons it not necessarily optimal, due to the reordering of data, which might be necessary. As a result composition languages such as SCL and $P^3L$ have been developed. These languages provide not only computation skeletons but also configuration skeletons specifying a particular data distribution.

As part of the NESL project a number of irregular algorithms have been implemented and their performance has been evaluated on machines such as a Cray-90 and a

CM-5. The largest of these algorithms are three versions of the n-body problem (Blelloch & Narlikar 1997), including the classical Barnes-Hut and the more recent Greengard algorithm, and a new parallel preconditioned conjugate gradient method for solving sparse linear systems of equations (Gremban et al. 1994). A set of parallel graph algorithms has been studied by Greiner (1994). All these examples use only data parallelism, which is supported in NESL via constructs similar to Haskell list comprehensions.

The Impala suite (Shaw 1998) is a collection of parallel programs mostly written in Id and SISAL. It is one of few publicly available packages of large parallel functional programs. Some performance results of the execution on parallel architectures such as Monsoon are included in the documentation of these programs.

## 4.10 Discussion

This chapter discussed an approach towards large-scale parallel lazy functional programming, which is based on a separation between algorithmic and behavioural code via evaluation strategies. With this technique Trinder et al. (1998) have gained wall-clock speedups for realistic programs over the most efficient sequential version of the program. Furthermore, the case studies in this chapter have demonstrated that the parallelisation and the performance tuning of a parallel program can often be done on top level, only changing the strategic part of the code and without the need to examine sub-modules in the code.

Evaluation strategies make heavy use of higher-order functions, polymorphism, laziness, and overloading. These features are very useful for achieving a high degree of modularity in sequential programs. They are of particular importance in the performance tuning of a parallel program where the evaluation degree may be specified in more detail in order to obtain good parallel performance. The examples in this chapter show that such tuning can be done in a data-oriented fashion, defining parallelism on intermediate data structures rather than in the modules and functions that create these data structures. This approach avoids breaking the abstraction provided by modules and functions and therefore enhances the modularity of the parallel program. This is demonstrated by the parallelisation of Lolita, which required changes in only two out of about three hundred modules to achieve a moderate amount of parallelism. The comparison of a pre-strategy version with a strategic version of LinSolv shows that the performance tuning of the parallel program is greatly facilitated by the modularity in the strategic parallel code. These examples show that lazy evaluation and parallel computation do not necessarily represent competing evaluation mechanisms but can be combined by explicitly defining parallelism, evaluation order, and evaluation degree on crucial data-structures in the program.

The comparison of the parallel functional programs with three parallel imperative programs, written in C, in Section 4.7 highlighted several important differences. In the imperative programs the lack of any separation between algorithmic and behavioural code required significant restructuring of the parallel algorithms to achieve good parallel performance. The lack of higher-order constructs did not allow the programmer to abstract commonly occurring patterns of computation in the same way as in functional languages. In particular, the parallelisation of LinSolv has shown the importance of parameterising strategies on complex data structures with strategies that should be applied to components of this data structure. Finally, the lack of algebraic data types in these languages resulted in rather clumsy code in processing the data and in handling the parallelism.

The performance tuning of the parallel programs discussed in this chapter emphasised the importance of several aspects of the dynamic behaviour of parallel programs. One of these aspects is the granularity of the program. The tuning of the Lolita system has shown that it can be useful to restrict the total amount of parallelism in the system in order avoid excessive use of resources. This has been done by using strategies to control the behaviour of the parallel program. The following chapters will discuss the aspect of granularity in more detail. This discussion aims at the development of runtime-system mechanisms that improve parallel performance by using granularity information.

# *Chapter 5*

# Granularity in Parallel Functional Programs

**Capsule**

The model of computation that is used in this thesis is explicit in *exposing* parallelism but implicit in *controlling* parallelism. A parallel program therefore describes what expressions may be evaluated in parallel and delegates decisions about how to coordinate the parallelism to the runtime-system. This can be seen as an intermediate step towards achieving fully implicit parallelism.

In controlling the parallelism in a parallel program many aspects have to be addressed: the synchronisation mechanism, the communication mechanism, the data locality during the computation, and the *granularity* of the computation. Intuitively granularity represents the amount of work that is available for each thread. Historically, many declarative languages that perform a naive implicit parallelisation, suffer from an extremely fine granularity. This increases total bookkeeping overheads such as thread creation time. The aim of a granularity control mechanism must therefore be to create only as many threads as are necessary to keep the machine utilised throughout the whole computation.

This chapter first discusses the problem of granularity in declarative languages in general. Then it proposes three concrete *granularity improvement mechanisms.* These mechanisms have been implemented in GranSim and an evaluation of their effectiveness is given. The chapter concludes by giving a comparison of these methods with other approaches for granularity improvement suggested in the literature.

## 5.1 Introduction

Extracting parallelism from a functional program is easy. In fact, it is so easy that even a compiler can do it. For example, strict arguments to a function can always be executed in parallel without changing the semantics of the program. Only the data dependencies in the program limit the degree of parallelism. However, this naive approach of parallelising a program is likely to achieve poor speedups because of the small number of computations in each parallel thread compared to a fixed overhead for starting the thread. The author's initial motivation for studying granularity came from experiences with an automatically parallelising compiler for a simple higher-order functional language, based on the dataflow model of computation (Loidl 1992). In this model every primitive operation is performed independently, imposing a huge synchronisation and thread creation overhead.

Over the years several approaches for improving the granularity in parallel functional languages have been proposed (Section 5.7 gives a detailed survey of these approaches). Most of these approaches are purely runtime-system based without any additional information about the program. Only rarely have concrete implementations with state-of-the-art optimising compilation used more than rather simple heuristics to achieve this goal. The approach advocated in this thesis is to combine a compile-time granularity analysis with runtime methods that use granularity information provided by the analysis to improve performance. This chapter concentrates on the runtime component of this system.

The structure of this chapter is as follows. Section 5.1 gives a general introduction and motivates the study of granularity. Section 5.2 discusses the control of parallelism in the runtime-system. Section 5.3 surveys the literature for studies on the importance of granularity. Section 5.4 analyses the impact of granularity on the performance of parallel programs in the eager-thread-creation and the evaluate-and-die model of evaluation. Section 5.5 proposes three granularity improvement mechanism. Section 5.6 presents measurements on parallel programs when using the granularity improvement mechanisms. Section 5.7 discusses related work and Section 5.8 summarises.

## 5.2 Dynamic Control of Parallelism

The art of constructing an efficient parallel program requires skills on many different levels. From a very abstract point of view one can distinguish between two different stages:

- exposing parallelism; and
- controlling parallelism.

In a parallel program written in GPH the potential parallelism is exposed via the placement of parallelism constructs by the programmer or by a system of automatic parallelisation. The previous chapter has shown how the parallelism can be explicitly controlled to some degree by using evaluation strategies. Typically, this more detailed description of the parallel program behaviour is added during the performance tuning of a parallel algorithm.

This chapter concentrates on runtime-system techniques for controlling parallelism. These techniques have the advantage of hiding low-level details of the parallel program behaviour from the programmer. In an idealised setting the runtime-system could make all low-level decisions. However, this requires a very flexible runtime-system, which can adapt to the characteristics of many different structures of parallelism. The studies in this chapter lead to the development of mechanisms that improve the flexibility of the runtime-system.

The classification of systems for parallel computation proposed by Sarkar (1989) distinguishes between the following major aspects of the language and the runtime-system:

1. exposing parallelism;
2. partitioning the program into threads, i.e. specifying sequential units of computation;
3. scheduling the threads on processors, i.e. specifying the mapping of parallel computations onto processors; and
4. communicating data.

In this classification, the model used in this thesis is one of explicit parallelism, implicit partitioning, implicit scheduling, and shared-memory communication. The first aspect of exposing parallelism has been discussed in detail in the previous chapter. Aspects 3 and 4 in this list are related to the notion of locality of threads and of data, which is hidden in the runtime-system. These issues of locality are important for an efficient parallel execution, but they are not the main topic of this thesis. This chapter focuses on the second aspect of the parallel execution, the partitioning of the program. In the model that is used in this thesis the partitioning of the program is performed dynamically. This chapter will study the effects of different evaluation models, eager-thread-creation and evaluate-and-die, on the dynamic partitioning of the parallel program. Based on these observations several mechanisms for reducing the overhead imposed by the parallel evaluation will be discussed. Finally, several measurements will assess the effectiveness of these mechanisms.

This notion of partitioning the program leads to the following notion of the granularity of the program.

**Definition 6 (granularity)** *The* granularity *of a parallel program is the average computation cost of a sequential unit of computation in the program.*

By this definition of granularity a parallel program is called fine-grained, if it consists of threads with only small pieces of computation compared to the total amount of computation. More informally the computation cost of a thread is sometimes called the "size" of the thread. Note that this definition based on computation cost excludes the overhead that is specific to the runtime system.

The creation and the management of parallel threads impose further costs on the execution of a program. For example, the creation of a thread requires operations like the allocation of a stack. In order to minimise this overhead it would be necessary to create only one thread — the program is executed sequentially. However, in order to achieve a good parallel performance a certain level of parallelism has to be maintained throughout the computation in order to make use of all available processors and to provide the possibility of overlapping computation and communication.

This tension between reducing parallelism overhead and maintaining a high degree of parallelism is illustrated by the graphs in Figure 5.1. The graph on the left hand side shows the total runtime of the program. The graph on the right hand side shows the parallelism overhead when executing the program. In both cases the x-axis represents

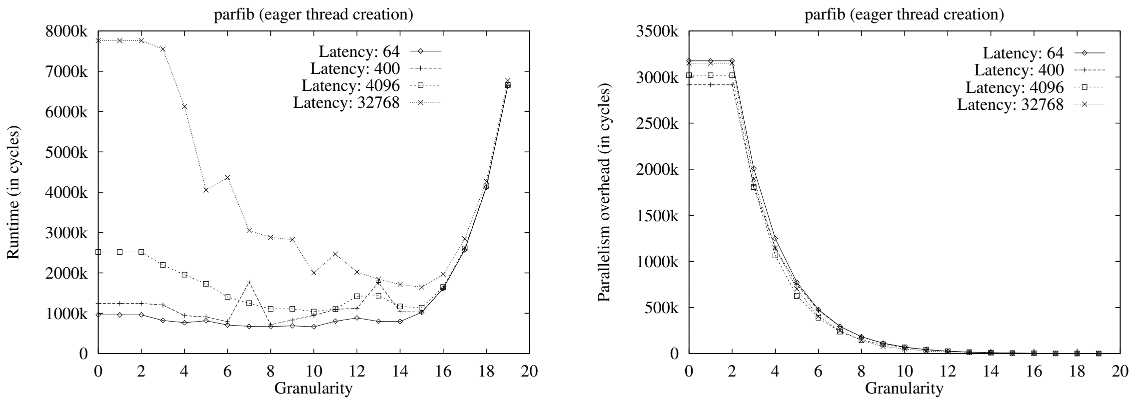


**Figure 5.1** Runtime and parallelism overhead with varying thread granularity

an increasing granularity of the program. In particular for a high latency setting the runtime drops together with a reduction in the parallelism overhead caused by a coarser granularity of the program. However, with increasing granularity the number of threads drops, too. The increase in runtime for coarse granularity is therefore caused by a lack of parallelism at some points during the computation ("starvation"). For decreasing latency the profiles of the runtime graph come closer to the idealised case where the shortest runtimes are achieved for extremely fine-grained programs consisting of a large number of threads.

Rather than showing the predicted behaviour of a parallel execution the graphs in Figure 5.1 have been obtained by running a parallel version of `nfib` under the GRANSIM simulator. Since `nfib` generates two threads in each recursive call it is a good, simple test case for studying the performance problems caused by massive parallelism with very small pieces of useful computation. Thus, runtime and latency are both measured in machine cycles. These measurements use a realistic setting for parallelism overhead and communication costs as it is imposed by the GUM system. The varying latency in the graphs represents a range of parallel machines from shared-memory machines, with low latency, to networks of workstations, with very high latency. The increase in granularity is obtained via a thresholding mechanism as it will be described in Section 5.5.1.

Concentrating on critical aspects of any parallel execution model it is possible to distinguish the following sources of parallelism overhead:

- Thread creation: this includes overhead for creating a thread descriptor and a stack for the thread;
- Synchronisation: in a shared-memory model this requires a check whether a closure has already been evaluated;
- Scheduling: thread descriptors have to be moved from the runnable queue to blocking queues and vice versa;
- Thread termination: usually old stacks are recycled, which bears some overhead but reduces thread creation overhead;
- Thread placement: if a thread is migrated to another processor the thread descriptor and the stack, or a part thereof, have to be sent to the new processor;
- Data placement: ideally, logically related pieces of data, such as the elements of a list, should be stored close to each other to avoid communication and make better use of the processor's cache.

One of the most important sources of parallelism overhead is the cost related to the creation of a new task. This requires the generation of a structure, a thread descriptor, that can hold the information about the current state of a task (in particular, the current register values) as well as the initialisation of a stack. One immediate consequence of this overhead is that the total time spent evaluating an expression should not be smaller than the cost of creating a thread because the latter is the minimum overhead attached to evaluating an expression in parallel.

## 5.3 Importance of Granularity

In the literature on runtime-systems for parallel functional programming several authors have examined the importance of granularity. This section gives a short survey, focusing on closely related work based on parallel graph reduction. A more detailed discussion, covering alternative approaches such as profiling and programmer annotations, can be found in Section 5.7.

Hammond et al. (1994) examine in detail the impact of different sparking strategies on the granularity and the runtime of a ray-tracer on the GRIP multi-processor.

Although, the emphasis is on data locality by avoiding to export a spark unless the global spark pool is low on work, the conclusions also highlight the importance of granularity in general. Among the three parallel versions that are studied the one with the coarsest granularity clearly performs best: in a 16 processor setup it is 8 times faster than a setup producing only small threads.

In his PhD thesis Goldberg (1988*a*) studies the efficiency of different strategies for creating parallel threads for arguments in a function application. He mainly considers the overhead for task creation, communication, on both ends, and cleaning up completed threads. He gives the outline of an exact, but infeasible, analysis of computation costs. Based on this analysis he develops a simple heuristics for estimating the computation costs of non-recursive program expressions. This information is used to partition a program into serial combinators. The program contains explicit constructs for creating and synchronising the execution of tasks. The body of each serial combinator is executed sequentially. Thus, the granularity of the program is directly expressed via the size of the serial combinators.

Goldberg observes that the optimal granularity depends on the architecture of the machine, especially its latency. In particular for high latency machines his results show that a coarse-grained computation performs better. His measurements also show that for shared memory machines the granularity of the threads is not very important. However, for distributed memory machines it is important, but the heuristics he gives are not generally strong enough to yield big improvements in performance. In particular, assigning infinite costs to all recursive functions loses too much information.

Maheshwari (1995) shows in the framework of the LAGER project (LArger Grain Graph Reduction) (Watson 1988) how the results of an asymptotic complexity analysis, although notoriously inaccurate, can still be used to improve the performance of a strict functional parallel program. Including the communication costs into the performance prediction proves to be an important issue. The main improvement in this work comes from determining an optimal schedule for generating parallel subprocesses based on user supplied cost information. The runtime-system methods used in this work rely on relative cost information in the form of priorities, which is somewhat similar to our priority mechanisms (see Section 5.5.2). However, in contrast to the work presented in this thesis, the LAGER project does not use an evaluate-and-die model of computation which dynamically increases granularity. This work does not address the question how to derive the cost information.

In summary, these methods emphasise the runtime aspect of improving the granularity of a parallel program. Heuristics for estimating computation costs have proven useful in increasing the granularity of parallel functional programs on real parallel machines. However, these methods are limited because they cannot derive the costs of recursive functions. The approach taken in this thesis, however, aims at a balance between a static granularity analysis that derives information automatically and runtime-system mechanisms that make use of this information.

## 5.4 The Relationship between Granularity and the Evaluation Model

The granularity of the program is of different importance for the different evaluation models discussed in Section 2.4.1. This section presents measurements that assess the importance of granularity in an eager-thread-creation model and an evaluate-and-die model.

### 5.4.1 Granularity with eager-thread-creation

In an eager-thread-creation model each potentially parallel expression is immediately turned into a thread. This simplest form of generating parallelism commits a thread to the evaluation of every expression that is annotated with a `par` construct. By making this choice very early no overhead for maintaining a spark pool is generated. However, this variant lacks the flexibility of dynamically increasing the granularity of a thread as it can be done in the evaluate-and-die mechanism. For this reason it is particularly important to avoid the generation of small threads in a model of eager-thread-creation.

Figure 5.2 illustrates the impact of the thread granularity on the speedup and the total number of threads for the `parfact` program. This simple divide-and-conquer program computes the sum of all integer values in a given interval by bisecting the interval in each stage. It is a very fine-grained program and is therefore a good test case for studying possible performance improvements with increasing granularity. In these

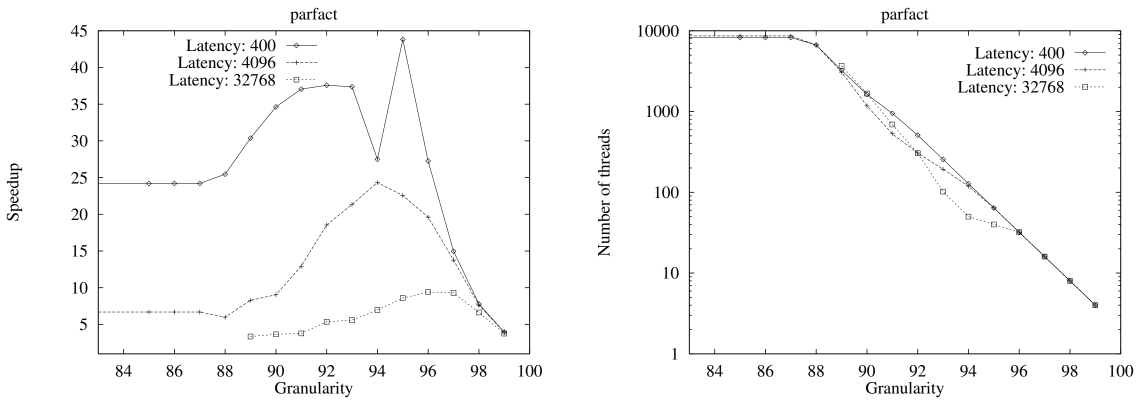


**Figure 5.2** Speedups and number of threads of `parfact` with eager-thread-creation

measurements the size of a thread is represented by the recursion depth in which it is generated. The root of the divide-and-conquer tree has level 100, the highest value, and this value is decremented in every recursive call. A granularity of e.g. 90 in the graphs in Figure 5.2 means that all sparks generated after more than 10 recursive calls are eliminated. This approximation of granularity aims at defining a simple relative ordering on the sizes of the generated threads, rather than representing an exact model of the computation costs for each of the threads. The measurements in this section have been obtained via GranSim using a realistic modelling of communication on a 64 processor machine.

The three graphs in Figure 5.2 represent a low, medium and high latency system. Even for very low latencies of 400 cycles a significant improvement in speedup with increasing granularity can be observed. In this case the speedup increases from 24.3 to 43.8, a factor of 1.8. For high latencies the absolute speedup is naturally smaller. The relative improvement in speedup, however, shows an even higher factor than for low latencies: from 3.3 to 9.3, a factor of 2.8.

The graph on the right hand side of Figure 5.2 shows the reduction in the number of threads with increasing granularity. Note that due to the use of a logarithmic scale this reduction is actually exponential. The small number of threads for very high granularities explains the drop in the speedup. For granularities higher than 95, which means that only the first 5 levels of recursion are used to generate parallelism, the total number of threads is smaller than the total number of processors in this

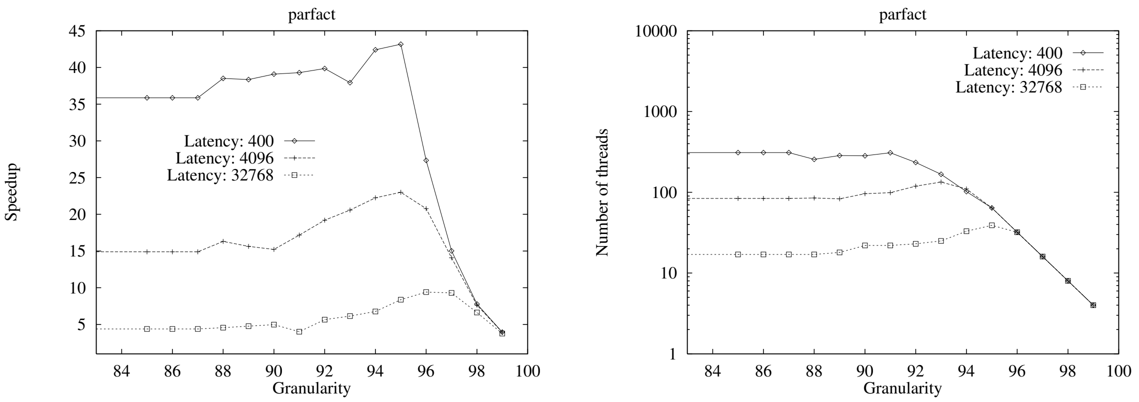


**Figure 5.3** Speedups and number of threads of `parfact` with evaluate-and-die

setup and starvation occurs.

### 5.4.2 Granularity with evaluate-and-die

In contrast to the eager-thread-creation mechanism in the previous section the *evaluate-and-die* mechanism dynamically increases the granularity in the system, similar to lazy task creation (Mohr et al. 1990). As discussed in Section 2.4.1 the evaluate-and-die model can subsume potential parallelism by allowing a thread to perform computations for which a spark has been created already. This requires the explicit management of a spark pool. In particular, in a divide-and-conquer structure it is possible that threads subsume sparks which represent child nodes in the computation tree.

Figure 5.3 shows the graphs for the same measurements as in Figure 5.2, this time using an evaluate-and-die evaluation mechanism. The direct comparison shows that the evaluate-and-die mechanism performs much better for small granularities: the finest-grained setup shows a speedup of 35.8 compared to 24.3 in the previous graph. This directly corresponds to a smaller number of threads generated with the evaluate-and-die mechanism: whereas the eager-thread-creation mechanism generates up to 8,245 threads, the evaluate-and-die mechanism does not create more than 310 threads. The overhead for managing these threads drops accordingly.

As a result of this better behaviour the performance improvement due to increased

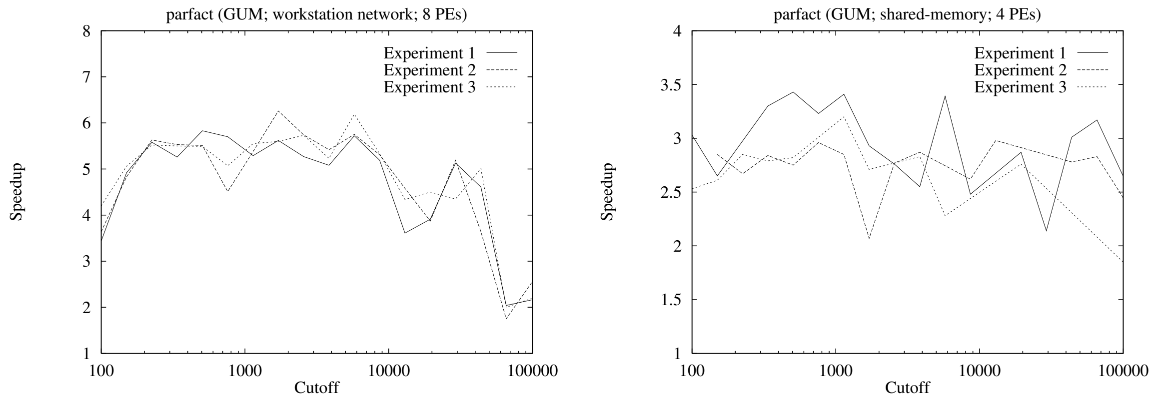


**Figure 5.4** Speedup of `parfact` (under GUM) on a workstation network and a shared-memory machine

granularity is far less pronounced for the evaluate-and-die mechanism. For a latency of 400 cycles the speedup increases from 35.8 to 43.1, a factor of 1.2. For a high latency of 32,768 cycles the speedup increases from 4.3 to 9.4, a factor of 2.2. It is interesting to note, however, that the speedup of the eager-thread-creation mechanism with optimal granularity is slightly higher than the best speedup obtained from an execution with an evaluate-and-die mechanism. This indicates that an eager-thread-creation mechanism can still outperform the evaluate-and-die mechanism, if it provides accurate granularity information to the runtime-system.

In order to relate these simulation results to the behaviour of `parfact` on a real parallel machine we have used GUM to run it on two parallel machines: a workstation network of 8 Suns 4/25 connected via ethernet with a rather high latency of circa 4 milliseconds for sending a packet of minimal size; and on a four processor SUN shared-memory machine. Figure 5.4 shows the speedups for 3 different experiments, with the workstation network results in the left hand graph and the shared-memory results in the right hand graph. Due to competing processes and general network traffic, these experiments show significant variations. However, the overall trend reflects the behaviour shown in the GranSim measurements. The improvements with increasing cut-off values are higher for the workstation network, in the best case the speedup increases from 3.64 to 6.26, a factor of 1.7. In comparison the shared-memory setup shows rather small and inconsistent improvements, in the best case the speedup increases from 2.53 to 3.20, a factor of 1.26. Overall, these results correspond to the GranSim results for a high latency and a low latency setup, respectively.

## 5.5 Granularity Improvement Mechanisms

This section discusses granularity improvement mechanisms (GIMs) that have been implemented in GranSim. Measurements for each of these mechanisms are given in the following section showing they can indeed improve the performance of some parallel programs.

In GranSim three granularity improvement mechanisms are available:

1. *Explicit threshold:* No spark whose priority is smaller than a given threshold will be turned into a thread. For this mechanism the user has to provide an explicit threshold value.

2. *Priority sparking:* The spark pool is treated as a priority queue with granularity information representing the priorities. This guarantees that the highest priority spark is turned into a thread. Priorities are not maintained for threads.

3. *Priority scheduling:* The thread pool is is treated as a priority queue with granularity information representing the priorities. This guarantees that the biggest available thread is scheduled next. This imposes a higher runtime overhead.

The motivation for investigating a threshold mechanism comes from the observation that such a mechanism is often used explicitly in the parallel code in order to increase the granularity of the generated threads. For example the `mergeStrategy` in Lolita (see Figure 4.11) encodes such a mechanism. Priority mechanisms, on the other hand, have proven useful in many applications in the area of operating systems. They provide a cheap way of accessing the best of a set of possible elements. By using such a generic data structure it is possible to profit from the research performed on optimising the operations on this data structure.

### 5.5.1 Explicit Threshold

The idea of this mechanism is to cut-off all sparks below a certain threshold. If the granularity information provided via the `parGlobal` or `parLocal` annotation is below a user supplied threshold level the spark will not be created at all. This represents the same kind of mechanism that is often used on a program level to control granularity. A typical idiom is

```
if arg < threshold
    then sequential code
    else parallel code
```

This style of programming has several drawbacks. The most important of these is the code duplication necessary to avoid repeated checks for the threshold. In contrast to the style advocated by evaluation strategies in Section 4.3, this code combines the algorithmic with the behavioural code. Therefore, the code becomes cluttered with conditionals that do not contribute to the definition of a value.

The explicit threshold mechanism provides runtime support for this style of programming. This means that the programmer does not have to plant a conditional statement into the code. Instead he can use the following piece of code:

```
parGlobal <name> <gran info> <size> <parallelism>
          parallel code
          continuation
```

The granularity information in this code is encoded via an integer value to minimise the overhead attached to it. It is used to provide information about the amount of computation required to evaluate the parallel code. This can be measured in evaluation steps or more abstractly by using this field as a priority. It is up to the programmer to take care that the computational complexity of evaluating the `<gran info>` field does not dominate the overall computation, which might outweigh the gain from avoiding the creation of many fine-grained threads.

The actual threshold value has to be provided as a parameter to the runtime-system. All potential sparks with a granularity information smaller than this value will not be created. This mechanism has been used in the measurements in the previous section to increase the granularity of the program.

### 5.5.2 Priority Sparking

When using a priority sparking mechanism the spark pool is treated as a priority queue with granularity information as priorities. This guarantees that the highest priority spark, i.e. the spark representing the largest piece of available work, is turned into a thread. Because small sparks remain in the spark pool for a long time, many of these sparks will be subsumed by other threads, increasing the total granularity of the program. In contrast to a priority scheduling mechanism priorities are not maintained for threads.

Priority queues are a fundamental data type with many applications in the areas of operating systems and parallel computation. The primary goal of this data structure is to provide cheap access to the "best" of a set of elements. To this end, the set is organised as a sorted sequence of elements. The key, by which the sequence is sorted, is called the priority. Typically, the following four operations are supported on priority queues:

- *findMin*, finding the minimum element of the queue;
- *insert*, inserting an element to the queue;
- *deleteMin*, discard the minimum element of the queue;
- *meld*, merging two priority queues;

For the GranSim runtime-system only the first three operations are needed. The implementation of priority queues exploits recent results of Brodal (1996) and Brodal & Okasaki (1996), which describe how to implement *findMin*, *insert* and *meld* in $O(1)$ and *deleteMin* in $O(\log n)$ time. The former algorithm is imperative, the latter is purely functional. These complexity functions are used in GranSim to simulate the costs of maintaining the priority queue.

One aspect specific to the use of queues in the GranSim runtime-system is the fact that sparks may be pruned. As soon as a closure has been evaluated, all sparks that have been generated for this closure may be pruned. In practice the pruning is not combined with an update because this would increase the costs in a common case in order to reduce costs for parallelism. However, sparks are pruned during garbage collection because the list of sparks has to be treated as a list of roots for the garbage collector and therefore has to be traversed anyway.

### 5.5.3 Priority Scheduling

When using a priority scheduling mechanism the thread pool is treated as a priority queue with granularity information as priorities. This guarantees that the biggest available thread is scheduled next. Maintaining granularity information on thread level imposes a higher runtime overhead. However, it allows the runtime-system to make better use of the available information. It also offers the possibility to dynamically adjust the priority of a thread based on other aspects of the dynamic behaviour. For example, in order to make use of good data locality, the priority of threads that rarely perform communication might be increased during the execution. Although, we have not studied mechanisms that dynamically change the priority of threads, it is an interesting possibility for extending this mechanism beyond the use of granularity information alone.

The handling of the priority queue for threads is the same as for sparks. In particular the same complexity functions for determining the costs of basic operations on the priority queue are used.

The effectiveness of any priority mechanism clearly depends on the number of elements from which the best element is chosen. Therefore, it should be noted that in combination with an evaluate-and-die model of computation the thread pools will be much smaller than the spark pools, because sparks are only turned into threads if there are no other runnable threads available on the current processor. On the other hand, the thread pool is updated very frequently, with every scheduling or descheduling of a thread. Therefore, compared to a priority sparking mechanism, a priority scheduling mechanism will more frequently choose from a smaller set of elements. The following measurements will assess whether the improved scheduling is worth the additional overhead imposed by this mechanism.

## 5.6 Using Granularity Improvement Mechanisms

This section focuses on possible improvements of the runtime when using the granularity improvement mechanisms discussed in the previous section. In general the priority mechanisms are more flexible than a simple thresholding mechanism because they retain granularity information rather than using it only to decide whether a spark should be generated or not. However, they also add additional overhead to the

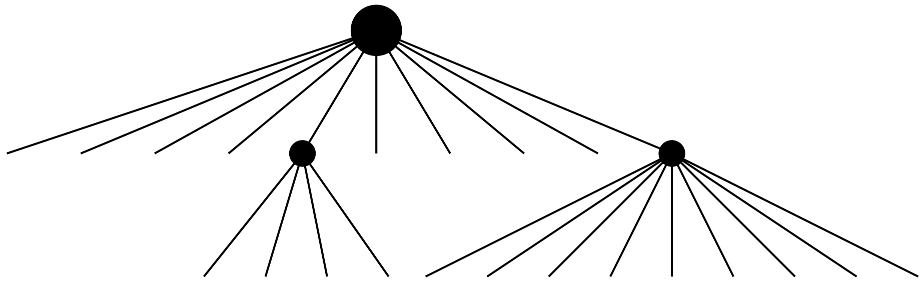

**Figure 5.5** Unbalanced divide-and-conquer tree generated by `unbal`

runtime-system as discussed in Section 5.5. All measurements in this section use an evaluate-and-die mechanism.

### 5.6.1 Divide-and-Conquer Programs

The `unbal` function shown below is a simple divide-and-conquer program that serves as an example of a computation where explicit granularity information can be used to improve the behaviour even with an evaluate-and-die mechanism. The function `parmap` in this code is a pre-strategy version of `parMap rnf`. Additionally, `parmap` takes as a first argument a cost function that computes a granularity estimate of applying the second function to a list element. In this case, the cost function returns the value 3 for all inputs resulting in a cheap computation. Otherwise the cost function returns the length of `list` as an approximation of the granularity.

The dynamic behaviour of this program can be represented as an unbalanced computation tree with decreasing sizes of computation. This structure is shown in Figure 5.5. Since the evaluate-and-die model only allows to subsume sparks generated for subtrees in such a structure, it cannot subsume all tiny threads in the tree, which might occur already as leaves close to the root of the tree.

```
unbal 0 = 1
unbal n
 | one_of_many n = n                              -- leaf case
 | one_of_few  n = maximum list                   -- node case
   where list = parmap costfn unbal [0..n-1]
```

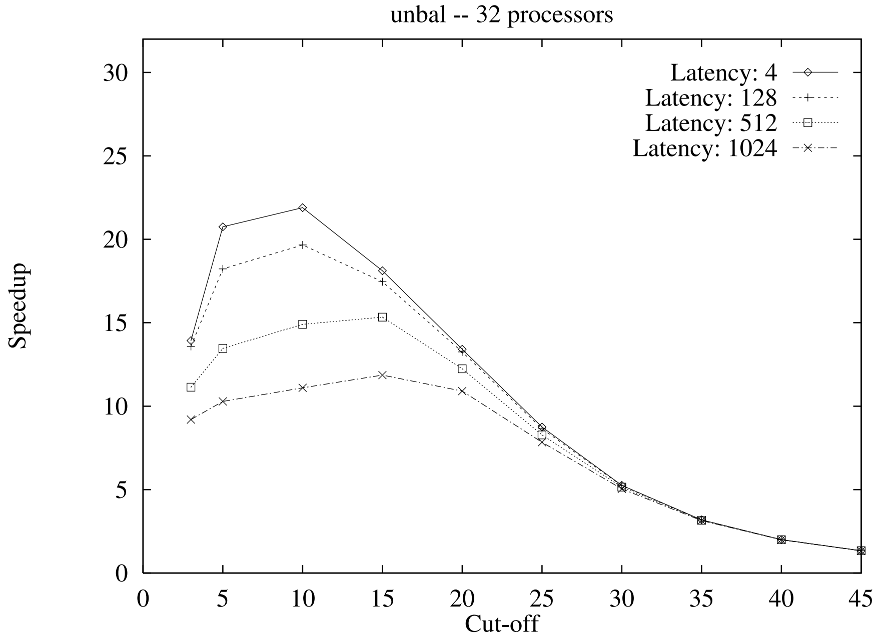


**Figure 5.6** Speedup of `unbal` with varying cut-off values

```
        costfn i = if one_of_many i then 3 else i
        one_of_few x = x `rem` diverge_every == 0
        one_of_many = not . one_of_few

diverge_every = 5
```

Figure 5.6 shows the speedup improvements when using a thresholding mechanism. The cut-off values are multiples of 5 because only every fifth node in the tree generates a large piece of computation (specified by `diverge_every` in the code above). Because of the unbalanced nature of the tree, which limits the effectiveness of spark subsumption, the improvements are much higher than the improvements shown in Section 5.4.2 for a balanced divide-and-conquer program.

The measurements in Figure 5.7 compare the relative runtimes and the absolute speedups of the program when using a priority sparking mechanism and a priority scheduling mechanism. The left hand graph graph shows the runtime relative to the runtime in a setup with no granularity improvement mechanisms (in percent). The priority mechanisms show a clear improvement for all latencies. However, the

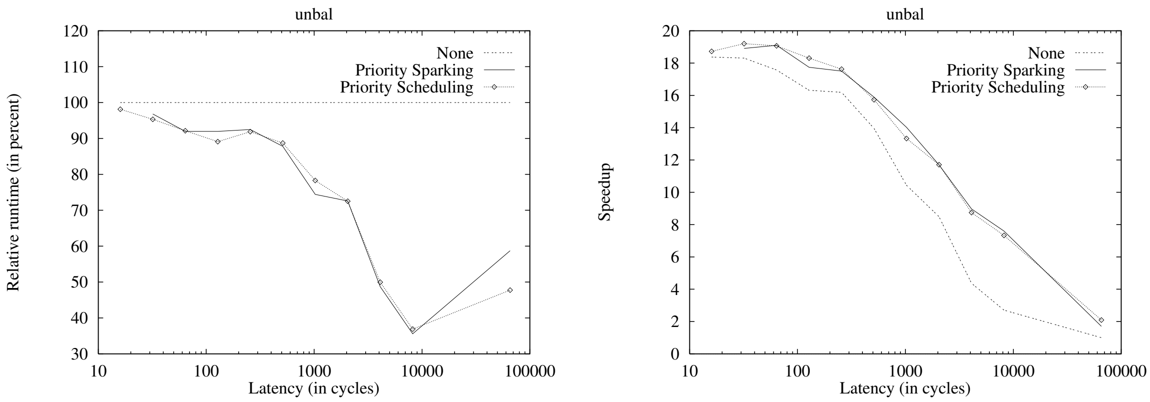


**Figure 5.7** Relative runtimes and speedups of `unbal` with priority sparking and scheduling

priority scheduling mechanism does not improve the runtime more than the priority sparking mechanism does. The main improvement comes from avoiding to generate tiny threads in the first place.

A more detailed assessment of the priority sparking and scheduling mechanisms for various divide-and-conquer programs is given in Loidl & Hammond (1995). In this paper it is shown that the improvement in runtime caused by these mechanisms directly corresponds to the average spark and thread queue lengths. For the priority queue mechanisms to be effective the dynamic behaviour of the program has to be such that these queues contain many elements to choose from. For example in the case of the `unbal` program with the measurements in Figure 5.7 the average spark queue length is 28 for latencies up to 256 cycles at the point of the best speedup. For programs with shorter average spark queue length the improvements in speedup are far less pronounced.

As a more realistic example program Figure 5.8 uses `queens`. This program finds all possibilities of placing 8 queens on a chess board without putting one of the queens in check. For most latencies the priority mechanisms yield a significant reduction in runtime. However, in a few cases the total runtime actually increases. This is to be expected, though, because the granularity information provided to the runtime system is not perfect. In this case the size of the board is used as a rough approximation to the granularity of the thread. With more accurate information generated by a static

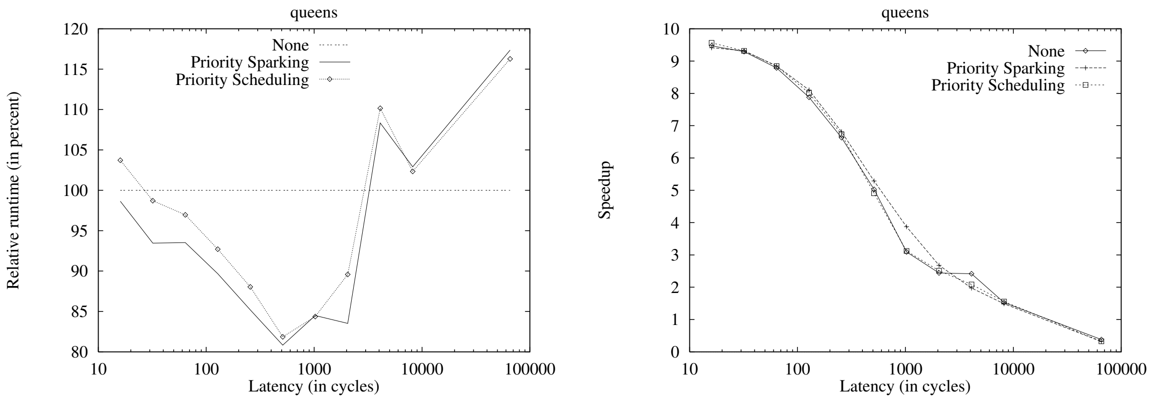


**Figure 5.8** Relative runtimes and speedups of `queens` with priority sparking and scheduling

analysis this information could be significantly improved. A more general problem is the lack of any information about the degree of parallelism in a thread. Without this information it may happen that a small thread that generates a lot of parallelism remains at the end of a long queue causing periods of low machine utilisation.

### 5.6.2 Larger Parallel Programs

Figure 5.9 studies the granularity of the parallel determinant computation with varying latencies using the priority queue mechanisms. For most latencies both a priority sparking and a priority scheduling mechanism manage to reduce the runtime compared to an ordinary parallel execution. The inverse priority sparking mechanism shown in the left hand graph represents the worst case scenario where the granularity information provided to the runtime-system is exactly inverse to the real computation costs. As a result the runtime may increase significantly in this setup. This behaviour indicates the danger of consistently providing wrong granularity information. Although the scheduling is hardly affected by occasional errors of the granularity information, consistent errors may lead to a serious degradation of the performance. The right hand side of this figure shows a clear reduction in parallelism overhead caused by thread creation and blocking threads when increasing the cut-off value in a thresholding mechanism.

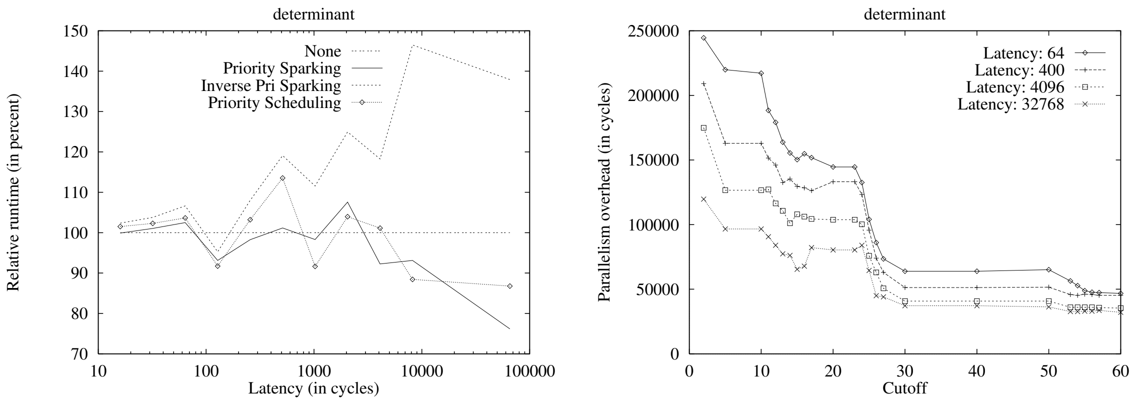


**Figure 5.9** Relative runtimes with variants of priority sparking and scheduling

The measurements in this chapter show that granularity improvement mechanisms can improve the efficiency of small parallel programs. This, of course, does not give clear evidence about possible improvements for large programs. However, in the performance tuning of the programs discussed in Chapter 4 it has been demonstrated that improving the granularity for large programs can be an important step in increasing the parallel performance. In particular, the final version of Lolita used an explicit thresholding strategy, discussed in Section 4.5. Similarly, we have used a generic granularity improvement strategy `parGranList`, discussed in Section 4.3.6, in the tuning of the bowing algorithm discussed in Hall et al. (1997). The main reason for not using the granularity improvement mechanisms developed in this chapter is the fact that they are currently only available in GranSim not in GUM. It would be natural to use a thresholding mechanism in the case of Lolita, and a priority sparking mechanism in the case of the bowing algorithm.

## 5.7 Related Work

Due to the importance of granularity for the efficient execution of parallel declarative languages, many attempts have been made to improve the granularity of the generated threads. This section gives a survey of the methods that focus on runtime control. Compile-time approaches for granularity improvement are discussed in Chapter 6.

Described on a more theoretical level than the work below, the controlled granularity algorithm of Aharoni et al. (1992) assumes that no knowledge of the size of a thread is available when deciding whether to create it. The main idea of this algorithm is that every thread performs an amount of work equal to the costs for creating a thread before itself creating another thread. This guarantees that in the worst case the parallel algorithm takes twice as long as a sequential algorithm. In contrast, the work presented in this chapter aims at improving the parallel runtime for different kinds of parallel programs rather than guaranteeing a certain worst case performance.

## 5.7.1 Runtime Methods

This section surveys runtime methods for increasing the granularity of functional programs. Additionally to the work that is discussed here in more detail several general systems have been designed to deal with fine-grained threads in an efficient way, e.g. the Cilk runtime system (Blumofe et al. 1995), the Cid system (Nikhil 1994, Nikhil 1995), StackThreads (Taura et al. 1994), and the filaments system (Lowenthal et al. 1996). Relationships to these approaches are outlined where appropriate.

### Load Based Inlining

One of the simplest methods for avoiding an abundance of parallel tasks is load-based inlining. In this approach the load of the machine is tested in order to decide whether a potentially parallel thread should be created or inlined, i.e. executed by the current task. The compiler has to generate two versions of the code: one for sequential and one for parallel execution. Load based inlining is used for example in the LAGER (Watson 1990) model, in the EQUALS system (Kaser et al. 1992), in GAML (Maranget 1991), in the Flagship machine (Keane 1994), and in the Cid (Nikhil 1994) parallel runtime system for symbolic computation. Although this model limits the total amount of parallelism, it has several severe problems:

- It is not possible to adapt to rapidly changing workload. Once a decision of creating or ignoring a spark has been made it cannot be rescinded even if the workload has changed in the meantime. This highlights the importance of delaying the decision whether to create or ignore a spark as long as possible. This fact has been observed by Sargeant (1991).

- If a child task is inlined and then blocks parallelism may be lost because the parent task is not necessarily blocked ("parent-child welding"). As Mohr et al. (1990) show for a simple prime number generator, inlining can even cause deadlock if one task blocks on another task that has been inlined by the same processor.
- Load-based inlining gives poor results for unbalanced computation trees and is ineffective for fine-grained linear recursions. However, Kranz et al. (1989) report good results for balanced trees.
- It is non-trivial to give a good threshold value for the workload of the machine that determines whether a task should be inlined.

### Lazy Task Creation

One of the most successful runtime approaches for improving granularity in a parallel system is *lazy task creation* (Mohr et al. 1990). The main idea in this approach is to create tasks only retroactively as processing resources become available. Thus, by default every task is inlined provisionally, but enough information is kept to selectively "un-inline" tasks. The programmer has to expose the parallelism in the program, e.g. with a `future` in Mul-T (Kranz et al. 1989). Overall the lazy task creation mechanism limits the total amount of parallelism that is generated. Starting from this idea, several variants of the basic mechanism have been studied:

**Continuation stealing.** This method was the first one used by Mohr et al. (1990) with lazy task creation. The basic idea is to distribute work by stealing continuations from the stack. To make this possible a "future queue" of pointers is maintained. Each entry in this queue points to a continuation in the stack. When stealing work the queue is traversed in a FIFO manner and, if available, a piece of work is stolen by copying the lower portion of the stack starting with that continuation, which will be turned into a parallel task. This method is almost identical to the sparking mechanism used in this thesis together with an evaluate-and-die model of computation. The main difference is that sparks are pointers to closures into the heap whereas the future queue contains pointers to the stack. The latter approach has slightly less bookkeeping overhead but the presence of an explicit spark pool makes it easier to attach additional information, such as granularity information, to a spark.

The main disadvantage of this approach is the runtime overhead that has to be paid:

- An explicit future queue has to be maintained. This has to be done even for a purely sequential execution to enable parallelism in later stages.
- When stealing work an unbounded portion of the stack has to be copied. This may cause a high amount of communication and the loss of data locality in the program.

**Lazy Threads.** In his thesis Rushall (1995) develops a new variant of lazy-task creation aiming at eliminating any bookkeeping overhead that is required during sequential execution to expose parallelism in later stages. The basic idea is to traverse the stack when stealing work and to make use of the continuation information available on the stack. This traversal of the stack is similar to the one required by a stop-and-copy garbage collector. It is very expensive but has to be done only when new work is needed, which means the overall system load is rather low.

In summary, Rushall's version of lazy-task creation, which is implemented on top of the G-machine using Haskell Core as the programming language, retrospectively transforms code

```
case (f x) of v1 -> case (g v1) of v2 -> ...
```

into

```
let z = g v1 in case (f x) of v1 -> case z of v2 -> ...
```

which means that a parallel thread can be generated for `z`. Note that in Haskell Core, a desugared version of Haskell that is used in GHC, the `case` expression forces the evaluation of the head expression. Therefore, nested `case` expressions enforce a specific evaluation order.

The stack is traversed in a FIFO order in the hope that older pieces of code represent larger evaluations. This is usually the case in balanced divide-and-conquer algorithms. Therefore, this model is particularly suited for this kind of algorithms. However, in the general case it cannot make use of granularity information because this information is not present any more when the parallelism is exposed. It would be possible to extend the model by inserting this kind of information on the stack, but this would add overhead in the common case and defeat the main advantage of this variant of lazy task creation.

Experiments with this form of lazy-task creation, implemented on a virtual shared-memory KSR1 machine, show that it is superior to sparking, as it used in this thesis, for very simple programs like `nfib`, where no closure has to be created in the sequential evaluation model but one is needed in order to create a spark. For bigger example programs, however, the difference is rather small. Lazy-task creation usually outperforms load-based inlining, although not consistently. In one example program, `iqueens+`, the sparking model gives better results because the lazy-task creation loses too much parallelism. In summary, for "well-behaved" divide-and-conquer programs the new lazy-task creation variant usually performs better than load-based inlining, but it is not a clear winner. For real applications the gap between the sparking model and the lazy-task creation model is rather small.

A similar approach is taken by Goldstein et al. (1996) in their work on *lazy threads*. They define a control hierarchy with varying overheads (sequential call, fork, and remote fork) and a storage hierarchy (stack, stacklet, and heap). This enables the compiler to pick the least expensive form of a function call and stack representation for a particular function call. An implementation of lazy threads in the Id90 compiler for the TAM machine achieves speedups of up to two over previous approaches of thread creation (Goldstein et al. 1996). It successfully uses fine-grained parallelism on a CM-5 distributed memory machine.

In his thesis Goldstein (1997) investigates the efficiency of different points in the control and storage hierarchies as well as different possibilities of thread representation and disconnection. The goal is to reduce the costs for thread creation and termination to little more than the costs of a sequential call and return. Disconnection decouples a lazy thread (spark) that has been turned into an independent thread from its parent thread. An eager disconnect scheme allows the parent to invoke children on its stack in exactly the same way as before, but bears a rather high overhead by copying a portion of the stack. In contrast, in a lazy disconnect scheme the child steals the stack from the parent and forces the parent to allocate new children on a new stack. This version avoids copying an activation frame even if a stack storage model is used. Two representations of potential parallelism, both planted in the stack, are investigated: continuations, as discussed above, and thread seeds. Thread seeds are basically pointers to code segments. They can be planted into the stack when a parallel ready sequential call is performed (implicit queueing), or the may be managed as an explicit queue similar to the future queue mentioned before.

The results of comparing different versions of thread representation (continuations and thread seeds) together with eager and lazy disconnection show that thread seeds with lazy disconnection perform best. Implicit queueing proves to be too expensive in creating a new parallel thread (a stack traversal is required). As a compromise a lazy queue is used, which starts as an implicit queue in the stack but is made explicit after the first steal request arrives. Goldstein concludes that "the performance is best in implementations that strike a balance between preparation before the potentially parallel call and extra work when parallelism is actually needed".

### Other Runtime Methods

In the framework of the Dutch Parallel Machine project (Barendregt et al. 1987), Hofman (1994) has developed runtime-system mechanisms for improving the granularity in the fork-and-join model of computation. In this project, a "sandwich" annotation is used to express parallelism: two phases of sequential strict evaluation flank one phase of parallel lazy evaluation. One problem in such a model of symmetric parallelism is potential gratuitous thread migration at the end of the computation, after merging the two parallel branches. The mechanisms developed by Hofman prevent threads to be moved to other processors after the join phase. This is based on the assumption that the amount of work after the join operation is rather small. This approach to parallelism is fundamentally different from the asymmetric parallelism obtained via the evaluate-and-die model: no synchronisation between child and parent task is enforced if the child finishes before the parent requires its result. Therefore the problem of a bottleneck at the end of the computation is less severe.

In the ZAPP project Burton & Sleep (1981) have developed an adaptive mechanism for throttling the parallelism in the system. Near the root of the computation tree a FIFO strategy (breadth first traversal) is used to create a high amount of parallelism. If the machine is sufficiently loaded a LIFO strategy (depth first traversal) is used to avoid excessive space consumption as well as creation of parallelism. This model has also been used as means of throttling the parallelism in the Manchester Dataflow machine (Ruggiero & Sargeant 1987).

$k$-bounded loops (Arvind & Nikhil 1990) in Id are used to limit the number of parallel threads, but the size of the threads is not automatically increased. The idea is to limit the number of loop bodies that may be executed in parallel to $k$. The main purpose is to reduce storage requirements. It has been shown that choosing the right $k$ value for $k$-bounded loops can improve performance dramatically (Culler 1990). However, so far no compiler controlled mechanism for finding good $k$ values has been developed and finding such values has proven to be quite hard in big applications such as an ocean modelling program (Shaw et al. 1996).

Rabhi & Manson (1990) present a hybrid method for improving task granularity in a parallel functional programming system. At compile time the parallel and the sequential complexity of a function are analysed. This information is used at run time to decide whether a computation is coarse-grained enough to be performed by a parallel task. In this paper especially divide-and-conquer programs are examined. This follows the approach of trying to detect common patterns in (recursive) cost expressions of function bodies in order to infer closed cost expressions. Some experimental results of that approach, mainly for divide-and-conquer programs, are presented in (Rabhi 1992).

### 5.7.2 Programmer Annotation Approaches

The most prominent work using this approach is Hudak's para-functional programming approach (Hudak 1986, Hudak 1991). This approach defines a set of annotations that control the creation and location of parallel tasks. The language issues have already been discussed in Section 4.9.1. The following discussion focuses on granularity issues.

The para-functional programming approach allows the programmer to have more or less direct control over the runtime system and thereby affect the granularity of the parallel tasks. For example the basic constructs in this approach make it possible to define serial combinators (Hudak & Goldberg 1985), which perform purely sequential computation without the need to synchronise. Therefore, the size of these combinators determines the granularity of the program and can be manipulated by the compiler. The most recent work in this area uses monads for obtaining system information, such as machine load, in a referentially transparent way (Mirani & Hudak 1995). The resulting language for scheduling and mapping computations is very flexible and close to evaluation strategies (Trinder et al. 1998) as discussed in Section 4.3. In particular functions can be parameterised with schedules describing their dynamic behaviour. This makes good use of the abstraction and the overloading mechanism in Haskell. Furthermore, stateful computation via monads is used to extract system information and to specify operational aspects used in the schedule for a parallel program. This system has been implemented on a Silicon Graphics 16 processor machine.

The Concurrent Clean system (Nöcker, Smetsers, van Eekelen & Plasmeijer 1991) also uses this approach. It defines a rich set of annotations that allows the programmer to change the reduction strategy of the system (van Eekelen & Plasmeijer 1993). By default it uses a lazy evaluation scheme. Two kinds of annotations are available: strict annotations, that locally force the use of eager evaluation, and process annotations, that determine the creation and placement of parallel tasks. The latter set of annotations is used for choosing the right level of granularity. Additionally, annotations for specifying graph sharing and copying are provided (Achten 1991).

Another parallel functional programming system that provides a rich set of annotations is the Hope$^+$ system for programming the Flagship parallel machine (Kewley & Glynn 1989). The strictness annotations enable the programmer to choose specific evaluators for expressions in the program. Dependency annotations control the evaluation order by describing how far a parameter in an expression has to be evaluated before starting the evaluation of the expression itself.

### 5.7.3 Profiling Methods

An alternative approach for extracting information about the granularity of generated tasks out of a program is to execute the program with some sample input and to generate profiling information. This information is then fed back into the compilation process and can be used to generate better (often coarser-grained) code. Sargeant obtained promising results using this approach on a virtually shared memory machine (Sargeant 1993). Sodan & Bock (1995) used this approach to obtain useful information specifically for granularity control on large programs. However, the main problem with this approach is the dependence on the choice of the initial, small input set. If the runtime behaviour of the program does not vary much between different inputs, then this approach will provide very good results without a large compile-time overhead. In general, however, the choice of good sample input is critical in this approach and it is not obvious, which metric to use to assess the quality of some input in this context.

This strategy can be very effective in combination with a skeleton-based approach to express parallelism. Algorithmic skeletons (Cole 1989) define the parallel behaviour of a set of higher-order functions, representing commonly occurring patterns of computation. By using a fixed set of well-studied functions it is easier to make statements about the dependence of the runtime behaviour on slightly different inputs. Busvine (1993) uses this approach in his implementation of the PUFF compiler. In a first step a the compiler exposes all parallelism down to the level of function calls via the insertion of `par` annotations. Then the program is run on one or more sets of data, collecting statistics about computation costs and execution frequencies. This information is used to transform the program into a parallel version that has increased granularity. A wide range of parallel programs generated with the PUFF compiler achieved good speedups on a distributed memory machine. In his PhD thesis Bratvold (1994) gives an overview of using skeletons for parallel programming. His results of combining a skeletons approach with profiling to gain information on granularity show good results on a distributed memory architecture. In particular he reports that the errors of profiling based performance prediction rarely exceed 20%. In contrast to Bratvold's system, which is specific to one parallel machine, Michaelson et al. (1997) present the design of an architecture-independent parallelising compiler for SML. It uses the same approach of structural operational semantics based instrumentation of the code in order to obtain granularity information via profiling. However, these costs are parameterised over machine specific parameters. Instantiating these parameters and combining the profiling information with expressions derived from the underlying cost model for skeletons should give accurate granularity information.

Darlington et al. (1995) have designed a structured coordination language SCL based on skeletons. In combination with Fortran as a computation language they report speedups of up to 70 on 100 processors on a distributed memory machine for a parallel matrix multiplication. A general treatment of the skeletons based approach has been provided by Rabhi (1995), who has related algorithmic skeletons to a number of parallel paradigms in designing a paradigm-oriented approach towards parallelism.

## 5.8 Discussion

This chapter has introduced the notion of granularity in parallel programs and motivated the importance of studying this particular aspect of parallel program behaviour. It has been demonstrated that granularity is more important for the performance of programs using an eager-thread-creation model. For a simple test program the speedup could be increased by a factor of 1.8 for low latency machines and by a factor of 2.8 for high latency machines. However, even for divide-and-conquer programs with an evaluate-and-die model it is possible to achieve performance improvements of a factor of 2.2 on high latency parallel machines. This has been shown via GranSim and GUM measurements. For unbalanced divide-and-conquer programs the possible performance improvement is even higher because the evaluate-and-die model is not able to subsume the same amount of gratuitous parallelism.

This chapter has presented three granularity improvement mechanisms:

- an explicit threshold mechanism,
- a priority sparking mechanism, and
- a priority scheduling mechanism.

In the measurements presented in this chapter the best results have been obtained with an explicit threshold mechanism. However, this mechanism assumes absolute granularity information. Such information is in general more expensive to produce than relative granularity information i.e. information that only allows to compare the granularities of two expressions in the program. Such relative granularity information is sufficient for the priority mechanisms. On the other hand, the priority mechanisms generate additional runtime overhead via the management of priority queues. Since the optimal cut-off value, which has to be provided explicitly to the runtime-system, is in general machine dependent it is not clear which mechanism will perform best for larger applications. However, having several such mechanisms available as part of the runtime-system gives the programmer additional flexibility in the performance tuning of a parallel program. Furthermore, the choice of the runtime-system mechanism for granularity improvement will in the end depend on the amount of information that can be automatically derived from the program. The next chapter will discuss this question by presenting a static granularity analysis for determining upper bounds of computation costs.

# *Chapter 6*

# Granularity Analysis

**Capsule**

Several examples in Chapter 5 have shown that information about the granularity of program expressions can be used by the parallel runtime-system to improve the performance of the program. This chapter discusses a granularity analysis for inferring an upper bound of the computational costs of an expression at compile-time. This analysis is a combination of two existing analyses, one for size and one for cost information. The granularity analysis is specified as an inference system for a strict higher-order language $\mathcal{L}$.

The inference system can only derive costs for non-recursive expressions. However, an extended cost reconstruction algorithm for this inference system is presented, which exposes recurrences over cost expressions in order to handle recursive functions. Thereby, the analysis can be combined with a library of recurrence relations and their known closed forms in order to generate a cost expression in closed form that depends only on the size of its input arguments. The chapter outlines an algorithm for a granularity analysis handling user defined recursion, based on the reconstruction algorithm presented here.

One of the major advantages of the chosen formulation of the granularity analysis as a type inference process over alternatives like abstract interpretation is its modularity. All relevant cost and size information of a function is attached to its type. Since all interface information required by the analysis is contained in the type, separate compilation and separate inference are possible. Finally, this chapter presents experimental results obtained from executing a program with attached granularity information. The cost inference has been done by hand in this case. The measurements show an improvement in speedup of more than 25% for eager-thread-creation and approximately 6% for evaluate-and-die.

## 6.1 Introduction

This chapter describes a granularity analysis for the simple strict higher-order functional language $\mathcal{L}$. The purpose of this analysis is to statically derive information about computation costs that can be used by the parallel runtime-system to improve performance, as discussed in Chapter 5.

The granularity analysis is presented as a type inference system. It derives information about the upper bounds of the size of data structures and the computation costs of expressions, provided the evaluation of the expression terminates. This analysis can be either seen as a part of a system for automatic parallelisation or as an off-line tool for the programmer to obtain additional information about the program's dynamic behaviour. However, the particular efficiency constraints of an on-line analysis and the details of its integration into the compiler are not discussed in detail here. The emphasis of this chapter lies on outlining an algorithm for performing the inference rather than proving its correctness.

The sized time system presented in this chapter is a combination of the inference system developed by Reistad & Gifford (1994) and sized types developed by Hughes et al. (1996). In particular, it also uses latent costs, which are attached to function types, in order to propagate cost information from function definitions to function applications in a higher-order language. Whereas the cost reconstruction algorithm by Reistad & Gifford (1994) only handles non-recursive expressions, our algorithm exposes cost and size recurrences from user defined recursive functions. This makes it possible to use a library of recurrences together with their closed forms in order to obtain cost information for some recursive functions. Such an approach has already been successfully used by Rosendahl (1986). The exact design of the library and the concrete formulation of the matching algorithm are further work.

The structure of this chapter is as follows. Section 6.2 discusses the main requirements for the analysis and the intended use of the derived information. Section 6.3 defines a small strict higher-order language $\mathcal{L}$. Section 6.4 presents the analysis as a sized time system. Section 6.5 describes the inference process including a size and cost reconstruction algorithm. This section also discusses how user-defined recursive functions are handled. Section 6.6 gives an example of an inference. A comparison of the presented approach with related work is given in Section 6.7. Finally, Section 6.8 summarises.

## 6.2 Design Philosophy

Before selecting a certain approach for performing a granularity analysis several design decisions have to be made. In particular, we have to address the following questions.

*How detailed does the cost information need to be?* This question has to be answered with respect to the runtime-system and its ability to use cost information. The measurements in Chapter 5 show that by using a thresholding mechanism, eliminating all small threads, significant performance improvements can be achieved. However, this mechanism requires absolute cost information. As an alternative to a thresholding mechanism a priority mechanism can be used. This requires only a relative ordering of threads, i.e. relative cost information. In order to make the usage of all granularity improvement mechanisms possible the presented analysis generates absolute cost information.

*How accurate does the information need to be?* An important observation about a granularity analysis is that the information can be wrong without causing error in computation. It is only used for optimisation without changing the semantics of the program. It can, however, cause an increase in runtime if wrong decisions are made very frequently. This has been shown by one of the measurements in Section 5.6.2. Note the difference to strictness analysis in which case wrong information can cause the program to fail where it should succeed.

*Should the analysis produce a lower or upper bound?* Lacking perfect information about runtime data, the analysis has to give an approximation of the real costs. Two possible choices are to infer a lower or an upper bound. Since the runtime-system uses granularity information to eliminate small threads, a lower bound seems to be the natural choice. However, this leads to inaccuracy when handling recursive functions since the lower bound would normally reflect only the base case. Previous measurements, however, have shown that treating all recursive functions equally does not yield satisfactory information (Goldberg 1988*a*). In order to avoid inaccuracy in the important case of recursive functions, the analysis presented here yields upper bounds. Furthermore, since the pure computation costs of evaluating an expression lazily are less than or equal to an evaluation in a strict language, the results of the analysis can also be used as an upper bound for the analysis of a lazy language. An alternative to this choice might be to devise a separate analysis that tries to infer information about the values in the head of a conditional. This could be used for common patterns of recursive functions, e.g. in testing whether the length of a list is zero.

*What kind of analysis technique should be used for the static analysis?* The main alternatives are abstract interpretation (Cousot & Cousot 1977) and type inference (Kuo & Mishra 1989). Abstract interpretation is well-studied and offers optimisations to make it more efficient. However, it has severe efficiency problems for higher-order functions. Thus, the motivation for choosing an inference based approach over an abstract interpretation based approach can be summarised as follows:

- Using type inference achieves *modularity* by attaching all relevant information of the analysis to the type of an expression. This fits naturally with separate compilation. In contrast, abstract interpretation always assumes a global view of the entire program, which clearly is problematic for large applications.

- It is hard to argue about the *quality of a result* obtained via abstract interpretation (Aiken et al. 1994). Choosing a more intuitive representation of terms over the abstract domain and using term rewriting to compute results may alleviate this problem (Seward 1995).

- By using a library of recurrences for eliminating derived recurrences in the domain equations it is possible to *tune the accuracy* of the results. Since granularity information is mainly of interest for optimisation and wrong information will never invalidate the semantics of the program, this is a very desirable feature in practice.

- Type inference, in contrast to abstract interpretation, does not require the abstract domain to be of finite height. Therefore it is possible to use positive integer values for costs and sizes.

*Is the information gained from the analysis of a strict language useful for a lazy language?* Clearly, the cost of evaluating an expression in a lazy language depends heavily on the demand on that expression. Therefore, a granularity analysis of a strict language will yield an upper bound for the cost of evaluating the same expression in a lazy language. Note that we only consider computation costs, but not the bookkeeping overhead related to eager or lazy evaluation. However, granularity information is mainly useful for rather small threads in an automatically parallelising system. Such a system needs strictness information in order to automatically expose parallelism over strict arguments. Therefore, the analysis would be only used on provably strict expressions, which justifies the design of an analysis for a strict language.

## 6.3 Syntax of $\mathcal{L}$

The language $\mathcal{L}$ is a very simple functional language, intended solely as a vehicle to explore static analysis for parallelism. $\mathcal{L}$ is strict, polymorphic, and higher-order with lists as its only compound data type.

The *abstract syntax* of $\mathcal{L}$ is given below. To simplify the presentation it is assumed that variables ($\mathtt{v} \in \mathit{Var}$) and constants ($\mathtt{k} \in \mathit{Const}$) are disjoint and that variable names in the program are unique. This avoids complications in the treatment of the assumption sets in the sized time system.

$$\begin{array}{rcl} \mathtt{e} & ::= & \mathtt{v} \mid \mathtt{k} \mid \lambda\mathtt{v}\ .\ \mathtt{e} \mid \mathtt{e}_1\ \mathtt{e}_2 \mid \mathtt{cons}\ \mathtt{e}_1\ \mathtt{e}_2 \mid \mathtt{null}\ \mathtt{e} \mid \mathtt{hd}\ \mathtt{e} \mid \mathtt{tl}\ \mathtt{e} \mid \\ & & \mathtt{letrec}\ \mathtt{v}\ =\ \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2 \mid \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 \end{array}$$

Overall, the structure of $\mathcal{L}$ expressions is similar to that of Lisp in that it focuses on lists as the only compound datatype. Local bindings via `letrec` are recursive. Since the entire $\mathcal{L}$ program is an $\mathcal{L}$ expression, nested `letrec` expressions have to be used to define auxiliary functions.

$\mathcal{L}$ uses sized types (Hughes et al. 1996): each type, except for the function type, has a component specifying an upper bound for its size. The type *Int* contains only positive integer numbers. Another extension to a conventional Hindley-Milner type system is the use of a cost expression in the function type, the latent cost, to propagate cost information from the function definition to its usage. The second annotation in the function type, $f$, represents a symbolic size function and is only needed for analysing recursive functions (see Figure 6.8). Finally, the construct $\beta^c$, a size pattern, represents a sized type with an upper bound of $c$, whose type component is unknown. Again, this construct is needed in the inference process to derive explicit recurrences for user defined recursive functions. In the following syntax of *type expressions*, $\alpha$ represents a sized type variable.

$$\tau \ ::= \ \alpha \mid \mathit{Int}^c \mid \mathit{Bool} \mid \mathit{List}^c\ \tau \mid \tau_1 \stackrel{c,f}{\to} \tau_2 \mid \beta^c$$

Both cost and size expressions are specified by *c-expressions*. Therefore, cost expressions can contain variables representing the size of a data structure. It is important to note that c-expressions are linear, i.e. there can be no expression of the form $v_1 * v_2$ where $v_1, v_2$ are variables. This property plays an important role in the implementation outlined in Section 6.5.

$$c \quad ::= \quad l \mid \omega \mid n \mid c_1 + c_2 \mid c_1 - c_2 \mid n * c \mid \max\ c_1\ c_2 \mid f\ c_1\ \ldots\ c_n \mid \text{sizeOf}\ \tau$$

In these c-expressions $n$ is an integer constant and $l$ is a c-variable. The $\omega$ symbol is used to express an unbounded cost/size. For sizes less than $\omega$ the operators $+$, $-$, $*$ and max behave as usual over integer values. When one of the operands is $\omega$ the result is $\omega$, too, with the exception of $x - \omega$ which is 0 for $x \neq \omega$ and $\omega$ otherwise. The $\leq$ relation, which will be introduced later, is defined as over integer values with $x \leq \omega$ for all $x$. In order to handle recursive programs symbolic cost functions $f$ have to be introduced. The arguments $c_1 \ldots c_n$ represent the sizes of the argument expressions in the program. The sizeOf construct is an auxiliary construct used to strip the size information from a sized type. Again, this is only necessary when deriving explicit recurrences describing cost and size for recursive functions (see Section 6.5.2).

Polymorphism is achieved in the usual way by quantifying over free variables of a `letrec`-bound expression. The use of sized types requires quantification over size as well as type variables. In the following $x$ is used to represent either a type or size variable. The general structure of *type schemes* is therefore:

$$\sigma \quad ::= \quad \forall x.\sigma \mid \tau$$

Note that sizes constitute parts of types in $\mathcal{L}$. This gives a convenient way to describe the size of sub-components of a data structure as well as the size of the structure itself, e.g.

$$\mathit{List}^5\ \mathit{Int}^{10}$$

denotes a list whose length is at most 5 with integer numbers no larger than 10 as elements. As an example of a type scheme, the type of the builtin constant `nil` is

$$\forall \alpha.\mathit{List}^0\ \alpha$$

## 6.4 A Static Cost Semantics for $\mathcal{L}$

This section develops a static cost semantics for $\mathcal{L}$. In order to statically estimate an upper bound for the cost of evaluating an expression, information about the size of values in the program is required. Therefore, a size analysis will be developed as well as a cost analysis. Both analyses are interwoven with a standard polymorphic type system to give a *sized time system* for $\mathcal{L}$. Because the size analysis and the cost analysis are presented in the same formal framework, this combination yields a concise description of the inference without repeating the same structure. However, it should be emphasised that this does not force an implementation of the analyses to use the same interwoven structure. The details of a possible implementation are discussed in Section 6.5.

### 6.4.1 A Sized Time System for $\mathcal{L}$

The inference rules of the sized time system in this section represent an extension to the standard type inference rules for $\mathcal{L}$, additionally inferring size and cost information. These extensions capture size and cost in a slightly different way. The size information represents a static property of a $\mathcal{L}$ expression and is therefore attached to its type. The cost information, however, represents a dynamic property of a $\mathcal{L}$ expression. It is therefore not attached to the type but inferred together with the size information. Cost inference uses size information but not vice versa.

The costs of higher-order functions are modelled by attaching *latent costs* (Reistad & Gifford 1994) to function types. These latent costs usually contain free variables representing the size of the arguments. This is illustrated by the following example. In order to derive the type for the expression $\lambda$ `f . ` $\lambda$ `x . f (x+1)` assume that the function `f` has type $\alpha \stackrel{c,f}{\rightarrow} \beta$. Here $c$ represents the latent costs of evaluating `f`. The annotation $f$ in the type can be ignored for this example. Then the type of the whole expression will be

$$(Int^{n+1} \stackrel{c,f}{\rightarrow} \alpha) \stackrel{0,f'}{\rightarrow} Int^{n} \stackrel{c+2,f''}{\rightarrow} \alpha$$

Thus, the cost of evaluating this abstraction is 2 steps plus the cost of evaluating the function `f`. In the system presented here costs are counted as steps. In order to improve the accuracy of the resulting cost expression it would be easy to use constants for basic operations like function expressions. However, we avoid such constants to make the structure of the inference clearer. In the above example, the costs are counted as one step for the + operation and one step for the function application. In general, the derived cost expression will depend on size variables in the argument types such as $n$. As a more detailed example, Section 6.4.2 gives the type of the function `length`.

Choosing step counts as computation costs also has the advantage of being high-level enough to abstract over the concrete computation model used in the implementation of the language. Thus, the analysis developed in this chapter is not tied to graph reduction. In order to tune the analysis to a specific model it would be necessary to assign basic costs for machine operations such as updating a closure in a graph reduction model or binding a value to a variable in an environment based model.

Figure 6.1 shows the extended type system. The c-expression in the superscript of a type is an upper bound for the size of the object. A judgement $\Gamma \vdash \mathtt{e} : \tau^z \ \$ \ c$ reads as follows: "Under the type assumptions $\Gamma$ the expression `e` has type $\tau$ (with size $z$) and a cost bound of $c$". The expression after \$ in a judgement is a c-expression that represents the cost for performing the corresponding computation. The assumption set $\Gamma$ contains bindings of variables, of constants, and of primitive operations to type schemes (of the form $x : \sigma$). Since all variable names are unique, assumption sets can be combined by using set union. The construct $\tau[x'/x]$ is used to denote a substitution of all free occurrences of $x$ in $\tau$ by $x'$. It extends to vectors, written as $\bar{y}_i$, by performing all substitutions simultaneously. The overall structure of the system uses inference rules resembling a Plotkin style structural operational semantics (Plotkin 1981) in a similar way to Tofte (1988).

The *(Var)* rule performs an instantiation of the abstracted size and type variables $x_i$ by substituting all free occurrences with fresh variables $y_i$ in the body of the type $\tau$. The $FV$ function computes the set of free variables in a type expression or an assumption set. For the inference of a program it is assumed that the initial environment contains mappings of variables representing basic operations like $+, *$ to their sized types. This avoids the necessity for an explicit rule on primitive operations.

The *(Weak)* rule allows to weaken, i.e. to relax, upper bounds for size and cost. It makes use of the subtyping relation $\trianglelefteq$ defined in Figure 6.2. Note that with this definition of $\trianglelefteq$, the relation $\tau_1 \trianglelefteq \tau_2$ alone does not imply that $List^{c_1}\ \tau_1 \trianglelefteq List^{c_2}\ \tau_2$, i.e. the subtype system is not structural. Because no subtype relations are defined

$$(Int)\ \frac{}{\Gamma \vdash \mathtt{n} : Int^n \ \$\ 0} \qquad (Bool)\ \frac{}{\Gamma \vdash \mathtt{b} : Bool\ \$\ 0}$$

$$(Var)\ \frac{}{\Gamma \cup \{\mathtt{v} : \forall \bar{x_i}.\tau\} \vdash \mathtt{v} : \tau'\ \$\ 0} \quad \begin{array}{c}\tau' = \tau[\bar{y_i}/\bar{x_i}]\\ y_i \notin FV(\tau) \cup FV(\Gamma)\end{array}$$

$$(Weak)\ \frac{\Gamma \vdash \mathtt{e} : \tau'\ \$\ c' \quad \tau' \trianglelefteq \tau \quad c' \leq c}{\Gamma \vdash \mathtt{e} : \tau\ \$\ c}$$

$$(Abstr)\ \frac{\Gamma \cup \{\mathtt{v} : \tau_1\} \vdash \mathtt{e} : \tau_2\ \$\ c}{\Gamma \vdash \lambda\mathtt{v}\ .\ \mathtt{e} : \tau_1 \overset{c,f}{\to} \tau_2\ \$\ 0}$$

$$(App)\ \frac{\Gamma \vdash \mathtt{e}_1 : \tau_1 \overset{c,f}{\to} \tau_2\ \$\ c_1 \quad \Gamma \vdash \mathtt{e}_2 : \tau_1\ \$\ c_2}{\Gamma \vdash \mathtt{e}_1\ \mathtt{e}_2 : \tau_2\ \$\ 1 + c_1 + c_2 + c}$$

$$(Cons)\ \frac{\Gamma \vdash \mathtt{e}_1 : \tau\ \$\ c_1 \quad \Gamma \vdash \mathtt{e}_2 : List^{c'}\ \tau\ \$\ c_2}{\Gamma \vdash \mathtt{cons}\ \mathtt{e}_1\ \mathtt{e}_2 : List^{c'+1}\ \tau\ \$\ 1 + c_1 + c_2}$$

$$(Null)\ \frac{\Gamma \vdash \mathtt{e} : List^{c'}\ \tau\ \$\ c}{\Gamma \vdash \mathtt{null}\ \mathtt{e} : Bool\ \$\ 1 + c}$$

$$(Hd)\ \frac{\Gamma \vdash \mathtt{e} : List^{c'}\ \tau\ \$\ c}{\Gamma \vdash \mathtt{hd}\ \mathtt{e} : \tau\ \$\ 1 + c}\ c' \geq 1 \quad (Tl)\ \frac{\Gamma \vdash \mathtt{e} : List^{c'}\ \tau\ \$\ c}{\Gamma \vdash \mathtt{tl}\ \mathtt{e} : List^{c'-1}\ \tau\ \$\ 1 + c}\ c' \geq 1$$

$$(Cond)\ \frac{\Gamma \vdash \mathtt{e}_1 : Bool\ \$\ c_1 \quad \Gamma \vdash \mathtt{e}_2 : \tau\ \$\ c_2 \quad \Gamma \vdash \mathtt{e}_3 : \tau\ \$\ c_3}{\Gamma \vdash \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 : \tau\ \$\ 1 + c_1 + (\max\ c_2\ c_3)}$$

$$(Letrec)\ \frac{\Gamma \cup \{\mathtt{v} : \tau_1\} \vdash \mathtt{e}_1 : \tau_1\ \$\ c_1 \quad \Gamma \cup \{\mathtt{v} : \forall \bar{x_i}.\tau_1\} \vdash \mathtt{e}_2 : \tau_2\ \$\ c_2}{\Gamma \vdash \mathtt{letrec}\ \mathtt{v} = \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2 : \tau_2\ \$\ c_1 + c_2} \quad \begin{array}{c}\bar{x_i} = FV(\tau_1)\backslash\\ FV(\Gamma)\end{array}$$

**Figure 6.1** A sized time system for $\mathcal{L}$

between basic types, this relation defines a set of inequalities over c-expressions alone.

The *(Abstr)* rule infers the cost of evaluating the body of a lambda-abstraction, and attaches this cost to the type of the lambda-abstraction as a *latent cost*. The latent cost usually contains a free variable for the size of the argument $x$. The symbolic size function $f$, which is attached to the function type, is only needed when a recursive function is defined. The handling of recursive functions is discussed in the reconstruction algorithm in Section 6.5.2. It is currently not reflected in the sized time system itself.

In the *(App)* rule the type of the function's domain must exactly match the type of the argument. Since types can be weakened by relaxing their size bounds this means that the size bound of the argument must be no greater than the size given in the type of the function's domain. The function application itself is counted as one step. Note that the latent cost $c$ of the function is added to the cost of the whole expression. In the case of a recursive function call, however, $c$ will be undefined at this point. Because the undefined $c$ will depend on the size of the argument $\mathtt{e}_2$, an explicit name $f$ is needed for the size function of $\mathtt{e}_1$. At the end of the inference, the cost expression for the recursive function will contain an application of the size function $f$ to the size of $\mathtt{e}_2$. Figure 6.8 shows the inference of the recursive function `length` as an example.

The rules for *(Cons), (Null), (Hd), (Tl)* show how size bounds are derived for list constructors and selectors. This system models a step counting semantics and therefore the application of a constructor to all of its arguments counts as one step. To increase accuracy it would be possible to add constants for the cost of these operations.

In general both branches of a conditional will have different sizes. The example below illustrates how the *(Weak)* rule is used to ensure that the types of both branches match as is required by the *(Cond)* rule. The cost bound of the conditional is the maximum of the costs of both branches plus the cost of the head of the conditional plus one step for performing the branch. In Section 6.5 we suggest some practical techniques for improving this cost bound.

The *(Letrec)* rule realises `letrec`-polymorphism as in the Hindley-Milner type system (Milner 1978). In the inference of a type for $\mathtt{e}_2$ the variable `v` is bound to a type scheme, which abstracts over type and size variables. Note that in the environment for typing $\mathtt{e}_1$ we do not use a type scheme for `v`, as in the Milner-Mycroft type system (Mycroft 1984), because this would make even plain type inference without sizes or costs undecidable, as shown by Henglein (1993). An instantiation of type schemes is performed as part of the *(Var)* rule. It is worth noting that the *(Letrec)* rule used by Hughes et al. (1996) is significantly more complicated because it has to propagate size information for algebraic data types from one recursion level to the next. In $\mathcal{L}$ this size propagation is encoded in the rules operating on lists. An extension of $\mathcal{L}$ to algebraic data types would have to add a size variable as an explicit iteration variable.

The reflexive and transitive subtyping relation in Figure 6.2 formalises the idea that

$$\frac{}{\tau \trianglelefteq \tau} \qquad \frac{\tau_1 \trianglelefteq \tau_2 \quad \tau_2 \trianglelefteq \tau_3}{\tau_1 \trianglelefteq \tau_3}$$

$$\frac{c_1 \leq c_2}{Int^{c_1} \trianglelefteq Int^{c_2}} \qquad \frac{c_1 \leq c_2 \quad \tau_1 \trianglelefteq \tau_2}{List^{c_1}\ \tau_1 \trianglelefteq List^{c_2}\ \tau_2} \qquad \frac{\tau_1 \trianglelefteq \tau_1' \quad \tau_2' \trianglelefteq \tau_2 \quad c' \leq c}{\tau_1' \xrightarrow{c',f} \tau_2' \trianglelefteq \tau_1 \xrightarrow{c,f} \tau_2}$$

**Figure 6.2** Subtyping relation for $\mathcal{L}$

the size component in a sized type specifies an upper bound. Therefore, it should always be possible to weaken this size bound. Similarly, the latent cost in a function type is an upper bound for the cost of evaluating the function. The need for such a subtyping relation can be motivated by an analysis of the following expression.

```
if (null xs) then 1 else 2
```

In this expression the `then` branch has a sized type of $Int^1$ but the `else` branch has the type $Int^2$. Only because of the subtyping relationship between these types $Int^1 \trianglelefteq Int^2$ is the above expression type correct. In the inference of this expression the *(Weak)* rule has to be applied to the `then` branch. The need for weakening latent costs can be motivated by observing that both sides of the conditional may yield a function type.

## 6.4.2 From Cost-Expressions to Cost-Functions

The sized time system in Figure 6.1 is a high-level description of how to infer costs and sizes of an expression. When deriving the cost of a function application the cost expression representing the latent cost for the function has to be used. However, if the function is recursive this approach will fail to yield a cost expressions in closed form because it has to refer to its own cost. Therefore, explicit names for unknown cost functions, symbolic cost functions, are needed. One symbolic cost function is needed for each recursive function in the program. In general the result of performing cost inference will be a set of recurrences that has to be solved separately. Section 6.5.4 discusses how this can be done.

The *(Letrec)* rule in the sized time system shows that the type schemes for `letrec`-bound functions in general contain universally quantified size variables. These variables are arguments to the cost function described by the inferred cost expression. In general we define for every function definition

```
f x1 ... xm = e
```

a cost function

$$f_{\mathcal{C}}\ l_1\ \ldots l_n\ = c$$

and a size function

$$f_{\mathcal{Z}}\ l_1\ \ldots l_n\ = z$$

where $c$ is the cost and $z$ is the size expression derived from `e`, the body of the function. The variables $l_1 \ldots l_n$ represent the size variables in the argument types of `f`. Note that $n$ depends on the type of the arguments to `f` because an argument of e.g. type $List^k\ Int^l$ will be translated into two size arguments $k$ and $l$. The cost reconstruction algorithm in Section 6.5.2 applies the function sizeOf on top-level in order to strip the size expressions from the resulting type.

One characteristic of the cost reconstruction algorithm discussed in Section 6.5.2 is the use of curried function application. This results in the introduction of separate cost functions in each function application. In order to obtain one cost function for the user defined function, as outlined above, it is necessary to merge these intermediate cost functions. This should be done in a separate simplification stage after cost reconstruction. The example in Section 6.6 discusses this point in more detail.

In summary, the sized time system assigns the following type to the polymorphic *length* function (Figure 6.8 describes the main steps of the inference, which is discussed in Section 6.5.2):

$$length : \forall\alpha.\ \forall l.List^l\ \alpha\ \stackrel{4*l+2,f}{\rightarrow}\ Int^l$$

In the following sections we use as a special notation $length_{\mathcal{Z}}$ for describing the corresponding size function ($length_{\mathcal{Z}}\ l\ =\ l$ in this case) and $length_{\mathcal{C}}$ for describing the corresponding cost function ($length_{\mathcal{C}}\ l\ =\ 4*l+2$ in this case).

## 6.5 Cost Inference

This section presents the outline of an algorithm for inferring upper bounds of size and cost in the presence of user defined recursive functions. In the case of non-recursive expression the sized time system presented in the previous section can be directly implemented using the same approach as Reistad & Gifford (1994). In order to handle user defined recursive functions, recurrences over an integer domain have to be constructed and solved. This section gives a reconstruction algorithm for exposing recurrences and proposes a general approach for solving the recurrences via a library.

An important feature of our subtype system is that all constraints range only over c-expressions rather than types in general. This can be seen from Figure 6.2, which defines $\trianglelefteq$ by adding only inequalities over c-expressions, but not over primitive types. In other words, from $\tau_1 \trianglelefteq \tau_2$ it follows that the Hindley-Milner types of $\tau_1$ and $\tau_2$ are the same, and this might subsequently be proved. Informally this can be shown by observing that omitting size and cost information from the sized time system yields the Hindley-Milner type system with additional rules for the basic list operations. The symbol $\vdash_{HM}$ is used to represent standard Hindley-Milner type inference.

**Conjecture 1** *Let* $\mathtt{e}_1$, $\mathtt{e}_2$ *be* $\mathcal{L}$ *expressions and* $\Gamma \vdash \mathtt{e}_1 : \tau_1$, $\Gamma \vdash \mathtt{e}_2 : \tau_2$, $\Gamma \vdash_{HM} \mathtt{e}_1 : \bar{\tau_1}$, $\Gamma \vdash_{HM} \mathtt{e}_2 : \bar{\tau_2}$. *Then*

$$\tau_1 \trianglelefteq \tau_2 \quad \text{implies} \quad \bar{\tau_1} = \bar{\tau_2}$$

It is important to note that no general subtype inference based on set inclusion is required, as it is done by Aiken et al. (1994) and Marlow & Wadler (1997). Instead, it suffices to solve inequality constraints over c-expressions, which range over integer values including infinity. Standard software packages exist for performing this test and the outlined inference algorithm uses such a package.

The structure of this section is as follows. Section 6.5.1 presents the overall structure of the inference process. Section 6.5.2 presents a size and cost reconstruction algorithm. This algorithm specifies a proof strategy for the sized time system, determining where to apply weakening and how to collect constraints. The result of the cost reconstruction algorithm is a sized type, a bound on the cost for evaluating the expression, and a constraint set of inequalities over c-expressions. The latter has to be solved separately. Section 6.5.3 defines a normal form on c-expressions. Section 6.5.4

addresses the question how to derive explicit symbolic cost functions for user defined functions and how to resolve a set of common recurrences. Finally, Section 6.5.5 addresses correctness issues of the presented inference.

### 6.5.1 Structure of the Inference

A *cost checking* algorithm for the sized time system is no more complicated than the existing size checking algorithm for sized types (Hughes et al. 1996). This algorithm uses the mandatory type declarations for all `letrec`-bound variables to compare the declared sizes with the sizes that are inferred from the body of the definition. This yields a set of inequalities over c-expressions in closed form.

Hughes' algorithm performs two separate passes for performing Hindley-Milner type inference and size inference, respectively. In maintaining this structure an additional pass for cost inference can be added. This pass would add inequalities over cost variables to the constraint set. Since both costs and sizes are represented via c-expressions, the same algorithm for collecting all inequalities can be used.

The satisfiability of the resulting constraint set can be checked by performing the Omega test (Pugh 1992). The Omega test is a state-of-the-art implementation of a decision test for the existence of integer solutions to affine constraints, which are a superset of the linear constraints as used in this thesis. If no solution exists the expression is ill-typed. Recursive functions do not pose any additional complication since their type has to be explicitly given. Such a checking algorithm could be used to confirm that a cost expression provided by the user, e.g. by hand analysing a function's complexity, is indeed an upper bound for the cost of the function.

When the checking algorithm is extended to *cost inference* a cost and size reconstruction algorithm has to handle functions of unknown type. This requires to add symbolic cost functions to the definition of c-expressions. These symbolic cost functions represent so far unknown cost functions applied to known size expressions. The reconstruction algorithm presented here will therefore extend the one developed by Reistad & Gifford (1994) by capturing the argument size of a function of unknown type in the *(App)* rule.

In order to solve the recurrences exposed by the reconstruction algorithm a "library" of recurrence relations with their closed form can be used. This is similar to the approach used by Rosendahl (1986). One basic difference is that the latter uses a sequence of source-to-source transformations in order to translate recursive step counting programs into non-recursive ones. In contrast, providing a library decouples the main part of the analysis from the recurrence elimination. Thus, the programmer has the possibility of adding recurrences to the library in order to improve the result of the analysis. In contrast to an abstract interpretation approach, this approach avoids the complexity of solving the resulting set of equations iteratively.

An open problem with an inference algorithm of this kind is how to find a minimal solution of the constraints that are derived. Since the plain type of the inference will be the same as the Hindley-Milner type, the plain type will be principal. However, if the "library" of recurrences contains approximations of closed forms the solution for costs and sizes will not be minimal. Adding such approximations has the benefit that unsolvable recurrences can be dealt with. Because the goal of the analysis is to derive some upper bound for the computation costs a minimal solution for the size component is not absolutely necessary in order to extract useful information out of the analysis. This agrees with observations by Reistad & Gifford (1994) on the quality of statically determined cost estimates.

In summary, the inference algorithm has the following global structure (see also Figure 6.3):

1. collect constraints, inequalities over c-expressions, while traversing the proof tree (see the cost reconstruction algorithm in Section 6.5.2);
2. simplify the set of inequalities, containing symbolic functions, by reducing c-expressions to a normal form (see Section 6.5.3);
3. spot common patterns of recurrences and replace them with closed forms, using a "library" of recurrences; if no matching recurrence is found the symbolic function is defined to yield $\omega$ for every input (see Section 6.5.4);
4. replace non-linear c-expressions with $\omega$; this step is needed as preparation for solving the constraint system using the Omega test;
5. eliminate trivial constraints containing $\omega$;
6. solve the resulting constraint system using the Omega test (Pugh 1992);

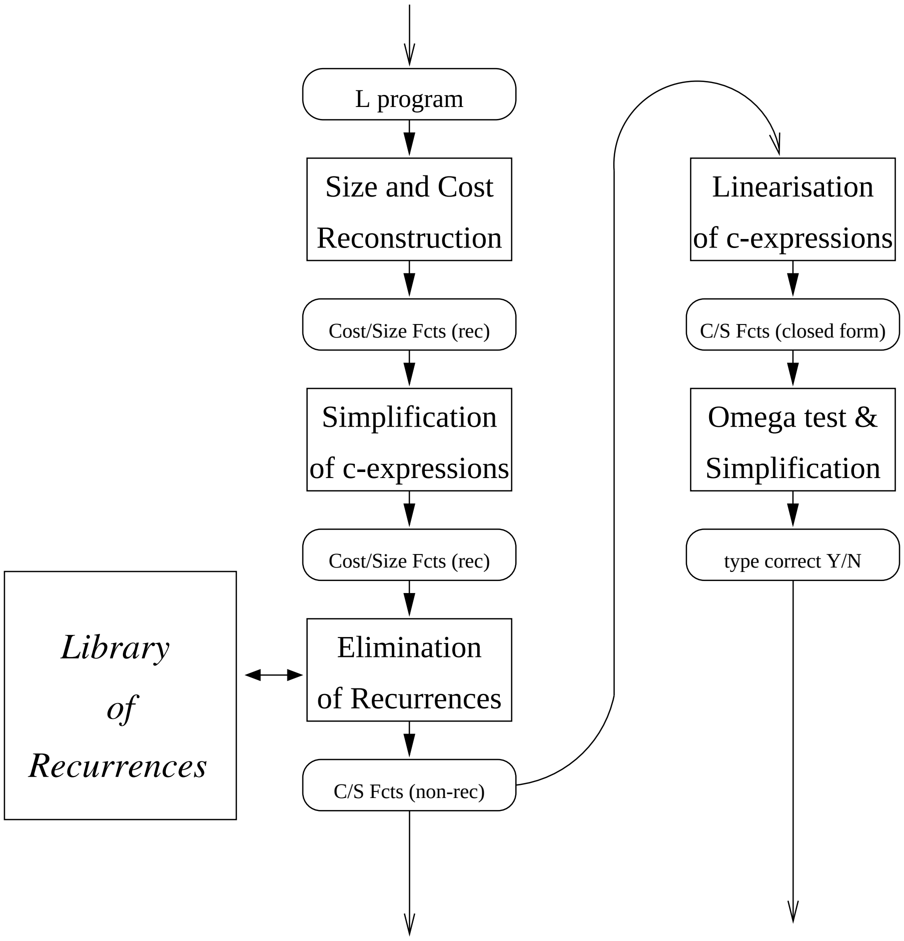


**Figure 6.3** Overall structure of the analysis

7. simplify the result further.

The main source of inaccuracy for the derived cost bounds in the sized time system presented here is the *(Cond)* rule. This might prove to be a problem if a costly branch is rarely executed, for example if the base case of a recursive function is much more expensive than the normal recursive case. Although this seems unlikely to be a major issue, one way to alleviate this particular problem would be to add special cases to the “library” of recurrences to avoid counting the base case several times. A variant of the max operator could then be used to indicate that the conditional is on the critical path of a recursion.

Another approach would be to extend the type system further by adding conditional types. Such types use runtime information to specify the type of an expression. In such a type system it is possible to formalise a dependence between the head and the branches of a conditional. For example, it is possible to type the expression

```
if (null xs)
  then 1
  else 1+length xs
```

as $(Int^1?List^0\ \alpha) \cup (Int^{1+n}?List^n\ \alpha)$. The conditional type constructor $\tau_1?\tau_2$ reads as "$\tau_1$ if $\tau_2$", incorporating runtime information into the type system without resorting to a safe approximation like the maximum of both sizes. Conditional types have proven useful in the optimisation of Lisp programs, where the aim is to avoid runtime type checking. Aiken et al. (1994) present a type system with union, intersection, and conditional types. A type inference algorithm has been implemented for FL and measurements on programs with several hundred lines of code show that typically no more than 10% of the compilation time is spent in the inference process. This shows that although conditional types require an extension to general intersection and union types, the resulting inference algorithm can still be fast enough to be usable in practice.

Another alternative for obtaining more accurate cost information for conditional expressions would be to use profiling information as additional input to the static analysis. Similarly to general profiling approaches discussed in Section 5.7.3, the program would be executed on a small sample input set before the static analysis is performed. The profiling stage would only have to collect data on the probability of taking each branch in the conditionals of the program. These probabilities could then be used in the static analysis as weights to the costs for both branches. This hybrid scheme would have the potential of combining the accuracy of a static analysis with the efficiency and additional runtime information of a profiling approach.

### 6.5.2 A Size and Cost Reconstruction Algorithm

The first phase in inferring computation costs is to reconstruct cost expressions from the program expressions. This frees the programmer of the burden of adding explicit types specifying size and cost bounds. Cost reconstruction requires a traversal of the inference tree and a bottom-up construction of costs. Since the sized time system uses subtyping the result is not only a sized type and a cost but also a set of constraints on size and cost variables.

### A Proof Strategy

A key question in designing a type reconstruction algorithm is where to apply the weakening rule. All other rules are structural and exhaustive. One possibility (Mitchell 1991) is to apply the weakening rule only at the leaves of the proof tree, i.e. immediately before a *(Var)* or one of the constant rules, *(Int)* and *(Bool)* in our case. The alternative, which is used by the cost reconstruction algorithm presented here, is to use the weakening rule only at the application rule (Hughes et al. 1996), in order to match the type of the argument with the type of the domain of the function, and at the conditional rule, in order to find a supertype of both branches.

### Algebraic Unification

In contrast to classical inference algorithms the unification algorithm used in this reconstruction algorithm returns a constraint set as well as a substitution. The syntactic structure of substitutions ($\theta$) and constraint sets ($C$) is as follows:

$$\begin{array}{rcl} \theta & ::= & \theta' \mid \theta'\theta \\ \theta' & ::= & \tau/\alpha \mid c/z \end{array} \qquad\qquad \begin{array}{l} C ::= C' \mid C'C \\ C' ::= c_1 \leq c_2 \mid c_1 = c_2 \end{array}$$

Note that substitutions are performed on types and sizes, whereas constraints only affect sizes (c-expressions contain only size but no type variables). The *algebraic unification algorithm* used in the following reconstruction algorithm is shown in Figure 6.4. It is inspired by the usage of algebraic unification in effect systems (Jouvelot & Gifford 1991). It implicitly applies the weakening rule wherever necessary by directly implementing the subtyping relation in Figure 6.2. It will, however, immediately fail if the shape of the two types is different. Because the constraint sets do not involve type variables this unification algorithm specialises to the Robinson's unification algorithm (Robinson 1965) if the size and type annotations are erased from all types. In this case the first component of the result, restricted to type expressions, is the substitution on plain types.

In Figure 6.4 $z, z_1, z_2$ denote c-variables, $\alpha$ denotes a type variable, $\beta$ denotes a type pattern, which ranges over a different name space than type variables, $c, c_1, c_2$ are c-expressions, and $\tau, \tau_1, \tau_2$ are type expressions. The cases for type variables, $\alpha$, and the type *Bool* are trivial. In the other cases the algebraic unification algorithm implements the subtyping relation by choosing upper bounds of the size annotations attached to the types. In the cases for *Int* and *List* the fresh size variable $z$ is defined to be an upper bound of $z_1$ and $z_2$. These relations are captured via inequalities in the constraint set, and therefore the algebraic unification algorithm has to return a constraint set as well as a substitution. In the case of function types an explicit substitution is used to guarantee that the names of the symbolic size functions are the same. As with standard unification the substitutions of nested types have to be composed. Additionally, the union of the constraint sets from the nested types has to be constructed, applying substitutions to propagate renamings into the constraint sets.

The unification on size patterns in the lower half of Figure 6.4 shows how size information is propagated even in the absence of plain type information by choosing an upper bound of the size expressions in the unified type expressions. Otherwise size patterns behave exactly like type variables. When unifying a function type with a size pattern (last but one rule) the size information has to be propagated through curried function application by choosing an upper bound for the sizes of both size patterns. Otherwise the size pattern behaves like a type variable. This case will be explained in more detail together with the *(App)* rule in Figure 6.5.

### The Reconstruction Algorithm

Figures 6.5 and 6.6 specify a size and cost reconstruction algorithm in the same inference style that has been used for the cost semantics of $\mathcal{L}$. For inferring the plain types the algorithm directly implements Milner's algorithm as presented in Field & Harrison (1988)[Chapter 7]. The additional rules for list operations are straightforward specialisations of the general application rule. The arguments to the algorithm are a type environment $\Gamma$ and the expression to be analysed. The result of the algorithm is a tuple $\langle \tau, \theta, c, C \rangle$, where $\tau$ is the sized type of the expression, $\theta$ is a substitution, $c$ is the cost of evaluating the expression, and $C$ is the constraint set, i.e. a set of inequalities over c-expressions. The constraint set plays the same role for size and cost variables as the substitution does for type variables. In the rules of

$$\mathcal{U} ::\ \ \tau \rightarrow \tau \rightarrow (\theta, C)$$

$$
\begin{array}{lcll}
\mathcal{U}(\alpha,\tau) & = & ([\tau/\alpha],\{\}) & \alpha \notin\ (FV(\tau)\backslash\alpha) \\
\mathcal{U}(Bool, Bool) & = & ([],\{\}) & \\
\mathcal{U}(Int^{z_1}, Int^{z_2}) & = & ([z/z_1, z/z_2], & z \text{ fresh} \\
 & & \ \{z_1 \leq z, z_2 \leq z\}) & \\
\mathcal{U}(List^{z_1}\ \tau_1, List^{z_2}\ \tau_2) & = & ([z/z_1, z/z_2]\theta, & z \text{ fresh} \\
 & & \ C \cup \{z_1 \leq z, z_2 \leq z\}) & (\theta, C) = \mathcal{U}(\tau_1,\tau_2) \\
\mathcal{U}\ (\tau_1 \stackrel{z_1,f}{\rightarrow} \tau_2, & = & ([z/z_1, z/z_1', f/f']\theta_2\theta_1, & z \text{ fresh} \\
\quad \tau_1' \stackrel{z_1',f'}{\rightarrow} \tau_2') & & \ \theta_2 C_1 \cup C_2 \cup \{z_1 \leq z, z_1' \leq z\}) & (\theta_1, C_1) \ = \ \mathcal{U}(\tau_1, \tau_1') \\
 & & & (\theta_2, C_2) \ = \ \mathcal{U}(\theta_1\tau_2, \theta_1\tau_2') \\
 & & & \\
\mathcal{U}(\alpha, \beta^z) & = & ([\beta^z/\alpha],\{\}) & \\
\mathcal{U}({\beta_1}^{z_1}, {\beta_2}^{z_2}) & = & ([\beta_2/\beta_1, z/z_1, z/z_2], & z \text{ fresh} \\
 & & \ \{z_1 \leq z, z_2 \leq z\}) & \\
\mathcal{U}(Bool, \beta^z) & = & ([Bool/\beta],\{\}) & \\
\mathcal{U}(Int^{z_1}, \beta^{z_2}) & = & ([Int/\beta, z/z_1, z/z_2], & z \text{ fresh} \\
 & & \ \{z_1 \leq z, z_2 \leq z\}) & \\
\mathcal{U}(List^{z_1}\ \tau, \beta^{z_2}) & = & ([List^z\ \tau/\beta], & z \text{ fresh} \\
 & & \ \{z_1 \leq z, z_2 \leq z\}) & \\
\mathcal{U}(\tau_1 \stackrel{z,f}{\rightarrow} {\beta_1}^{z_1}, {\beta_2}^{z_2}) & = & ([\tau_1 \stackrel{z,f}{\rightarrow} {\beta_1}^{z}/\beta_2], & z \text{ fresh} \\
 & & \ \{z_1 \leq z, z_2 \leq z\}) & \\
\mathcal{U}(\tau_1 \stackrel{z,f}{\rightarrow} \tau_2, \beta^{z_1}) & = & ([\tau_1 \stackrel{z,f}{\rightarrow} \tau_2/\beta],\{\}) &
\end{array}
$$

The symmetric cases for size patterns and size variables are omitted

**Figure 6.4** An algebraic unification algorithm on sized types

the reconstruction algorithm $z, l$ denote c-variables, where $l$ is used to represent the length of lists. $\alpha$ denotes a type variable, $\beta$ denotes a size pattern. $f_c, f_z$ denote a symbolic cost and size functions.

The proposed algorithm is based on the one developed by Reistad & Gifford (1994) for the cost reconstruction of FX programs. This algorithm traverses a given program expression and reconstructs its type, cost, substitution, and a constraint set over size and cost variables. This is the same quadrupel the algorithm in Figures 6.5 and 6.6 is using, and thus the combination of constraint sets and the composition of

$$(Int)\ \frac{}{\Gamma \vdash \mathtt{n} \ :\ \langle Int^{z}, [], 0, \{z = n\}\rangle}\ z \text{ fresh} \qquad (Bool)\ \frac{}{\Gamma \vdash \mathtt{b}\ :\ \langle Bool, [], 0, \{\}\rangle}$$

$$(Var)\ \frac{}{\Gamma \cup \{\mathtt{v} : \forall \bar{x_i}.(\tau, C)\} \vdash \mathtt{v}\ :\ \langle \theta\tau, [], 0, \theta C\rangle}\ \begin{array}{l}\bar{y_i} \text{ fresh}\\ \theta = [\bar{y_i}/\bar{x_i}]\end{array}$$

$$(Abstr)\ \frac{\Gamma \cup \{\mathtt{v} : \alpha\} \vdash \mathtt{e}\ :\ \langle \tau, \theta, c, C\rangle}{\Gamma \vdash \lambda\mathtt{v}\ .\ \mathtt{e}\ :\ \langle \theta\alpha \overset{z,f}{\rightarrow} \tau, \theta, 0, C \cup \{z = c\}\rangle}\ \alpha, f, z \text{ fresh}$$

$$(App)\ \frac{\begin{array}{c}\Gamma \vdash \mathtt{e_1}\ :\ \langle \tau_1, \theta_1, c_1, C_1\rangle \qquad \theta_1\Gamma \vdash \mathtt{e_2}\ :\ \langle \tau_2, \theta_2, c_2, C_2\rangle\\ (\theta, C) = \mathcal{U}(\theta_2\tau_1, \tau_2 \overset{f_c, f_z}{\rightarrow} \beta^{z})\end{array}}{\begin{array}{ll}\Gamma \vdash \mathtt{e_1}\ \mathtt{e_2}\ : & \langle\ \theta\beta^{z_1},\\ & \theta\theta_2\theta_1,\\ & \theta(1 + \theta_2\theta z_2 + \theta_2 c_1 + c_2),\\ & \theta(\theta_2 C_1 \cup C_2 \cup C \cup\\ & \{z_1 = (f_z\ z), z_2 = (f_c\ z), z = \text{sizeOf}(\theta\tau_2)\})\rangle\end{array}}\ \begin{array}{c}f_c, f_z, \beta,\\ z, z_1, z_2\\ \text{fresh}\end{array}$$

**Figure 6.5** A size and cost reconstruction algorithm for $\mathcal{L}$

substitutions are very similar. In contrast to the sized time system, however, the type system for FX does not use a separate weakening rule. Instead, the subtyping relation, which corresponds to the $\trianglelefteq$ relation in Figure 6.2, is combined with the remaining rules. Furthermore, the cost reconstruction algorithm in Reistad & Gifford (1994) uses a simpler unification algorithm, which only unifies annotated types producing a substitution but no constraint set. Instead, the reconstruction algorithm for FX adds inequality constraints on costs and sizes in the leaves of the inference tree. The algorithm in Reistad & Gifford (1994) is based on reconstruction algorithms in effect systems (Lucassen & Gifford 1988, Talpin & Jouvelot 1992, Debbabi et al. 1997), which extend type systems in order to capture information about side-effects in impure functional programs.

The algorithm presented here extends the cost reconstruction algorithm for FX by exposing recurrences over c-expressions describing cost and size of user defined recursive functions. This is done by attaching symbolic size functions to the function type and stripping the size information from an inferred type via the sizeOf function in the *(App)* rule. A *size pattern* of the form $\beta^{c}$ propagates information about the size of the argument through a function application even if the type of the result is unknown. Without this construct recursive functions would generate recurrences like $z = z + 1$

$$(Cons)\quad \frac{\begin{array}{c}\Gamma \vdash \texttt{e}_1 : \langle \tau_1, \theta_1, c_1, C_1\rangle \qquad \theta_1\Gamma \vdash \texttt{e}_2 : \langle \tau_2, \theta_2, c_2, C_2\rangle \\ (\theta, C) = \mathcal{U}(\tau_2, List^l\ \theta_2\tau_1)\end{array}}{\begin{array}{ll}\Gamma \vdash \texttt{cons}\ \texttt{e}_1\ \texttt{e}_2 : & \langle List^z\ \theta\theta_2\tau_1, \\ & \theta\theta_2\theta_1, \\ & \theta(1+\theta_2 c_1 + c_2), \\ & \theta(\theta_2 C_1 \cup C_2) \cup C \cup \{z = \theta l + 1\}\rangle\end{array}}\quad z, l \text{ fresh}$$

$$(Null)\quad \frac{\Gamma \vdash \texttt{e} : \langle \tau, \theta, c, C\rangle \qquad (\theta', C') = \mathcal{U}(\tau, List^l\ \alpha)}{\Gamma \vdash \texttt{null e} : \langle Bool, \theta'\theta, 1+\theta' c, \theta' C \cup C'\rangle}\quad \alpha, l \text{ fresh}$$

$$(Hd)\quad \frac{\Gamma \vdash \texttt{e} : \langle \tau, \theta, c, C\rangle \qquad (\theta', C') = \mathcal{U}(\tau, List^l\ \alpha)}{\Gamma \vdash \texttt{hd e} : \langle \theta'\alpha, \theta'\theta, 1+\theta' c, \theta'(C \cup \{l \geq 1\}) \cup C'\rangle}\quad \alpha, l \text{ fresh}$$

$$(Tl)\quad \frac{\Gamma \vdash \texttt{e} : \langle \tau, \theta, c, C\rangle \qquad (\theta', C') = \mathcal{U}(\tau, List^l\ \alpha)}{\Gamma \vdash \texttt{tl e} : \langle List^z\ \theta'\alpha, \theta'\theta, 1+\theta' c, \theta'(C \cup \{l \geq 1\}) \cup C' \cup \{z = \theta' l - 1\}\rangle}\quad \begin{array}{c} z, \alpha, l \\ \text{fresh}\end{array}$$

$$(Cond)\quad \frac{\begin{array}{c}\Gamma \vdash \texttt{e}_1 : \langle \tau_1, \theta_1, c_1, C_1\rangle \\ \theta_1\Gamma \vdash \texttt{e}_2 : \langle \tau_2, \theta_2, c_2, C_2\rangle \qquad \theta_2\theta_1\Gamma \vdash \texttt{e}_3 : \langle \tau_3, \theta_3, c_3, C_3\rangle \\ (\theta, C) = \mathcal{U}(\theta_3\theta_2\tau_1, Bool) \qquad (\theta', C') = \mathcal{U}(\theta\theta_3\tau_2, \theta\tau_3)\end{array}}{\begin{array}{ll}\Gamma \vdash \texttt{if}\ \texttt{e}_1\ \texttt{then}\ \texttt{e}_2\ \texttt{else}\ \texttt{e}_3 : & \langle\ \theta'\theta\tau_3, \\ & \theta'\theta\theta_3\theta_2\theta_1, \\ & \theta'\theta(1 + \theta_3\theta_2 c_1 + \max\ (\theta_3 c_2)\ c_3), \\ & \theta'\theta(\theta_3\theta_2 C_1 \cup \theta_3 C_2 \cup C_3 \cup C' \cup C))\rangle\end{array}}$$

$$(Letrec)\quad \frac{\begin{array}{c}\Gamma \cup \{\texttt{v} : \alpha\} \vdash \texttt{e}_1 : \langle \tau_1, \theta_1, c_1, C_1\rangle \\ \theta_1\Gamma \cup \{\texttt{v} : \forall \bar{x_i}.(\tau_1, C_1)\} \vdash \texttt{e}_2 : \langle \tau_2, \theta_2, c_2, C_2\rangle\end{array}}{\Gamma \vdash \texttt{letrec v} = \texttt{e}_1\ \texttt{in}\ \texttt{e}_2 : \langle \tau_2,\ \theta_2\theta_1,\ \theta_2 c_1 + c_2,\ \theta_2 C_1 \cup C_2\rangle}\quad \begin{array}{c}\alpha \text{ fresh} \\ \bar{x_i} = (FV(\theta_1\tau_1) \\ \cup FV(C_1)) \\ \setminus FV(\theta_1\Gamma)\end{array}$$

**Figure 6.6** A size and cost reconstruction algorithm for $\mathcal{L}$ (continued)

over c-variables because the information about the size of the argument in a recursive call is lost. The only solution to this equation is $\omega$, which would assign infinite costs to all recursive expressions. In the inference of `length` in Figure 6.8 this can be observed in branch 3, where the size of the expressions `tl xs`, namely $List^{l-1}\ \alpha'$, has to be propagated through the so far unknown type of the function `length`. In simple Hindley-Milner type inference no such information has to be propagated, because the Hindley-Milner types of `xs` and `tl xs` are the same.

As the *(Int)* case indicates, the algorithm maintains the invariant that size annotations in sized types are always variables. Thus, an explicit constraint $z = n$ has to be added

$$\begin{array}{lcl}
\text{sizeOf} :: \tau \rightarrow [c] & & \\
& & \\
\text{sizeOf}(\alpha) & = & [\,] \\
\text{sizeOf}(\beta^z) & = & z \\
\text{sizeOf}(Bool) & = & [\,] \\
\text{sizeOf}(Int^z) & = & z \\
\text{sizeOf}(List^z\ \tau) & = & z : \text{sizeOf}(\tau) \\
\text{sizeOf}(\tau_1 \overset{z,f}{\rightarrow} \tau_2) & = & f
\end{array}$$

**Figure 6.7** Definition of size stripping

to the constraint set in the *(Int)* case rather than just using $Int^n$ as in the sized time system. This invariant simplifies the algebraic unification algorithm.

The *(Abstr)* rule adds fresh variables for the cost and the size function attached to the derived function type. The constraint set captures the costs of evaluating the body of the function.

The *(App)* rule shows how the size information is propagated through a function application. The unification of the type of $\mathtt{e}_1$, $\tau_1$, with an explicit function type is standard. However, rather than choosing a type variable in the codomain a size pattern $\beta^z$ is used. Together with the algebraic unification algorithm this guarantees that the size information is not lost if the plain type of the result is unknown. Note that size patterns are only introduced in the codomain of function types. In the case of curried function applications, yielding size patterns, the result of the first application will be unified with another function type. The rule for unifying function types with size patterns in Figure 6.4 ensures that the size on the final result is an upper bound over all collected size information. The size component in the size pattern has the form $f_z$ sizeOf$(\theta\tau_2)$. The sizeOf function is used to strip size information from the sized type and to make it explicit in the application of the size function $f_z$. Figure 6.7 shows the definition of sizeOf, which will be applied at top-level when generating cost and size functions from the generated cost and size expressions. The result of applying sizeOf is a list of cost expressions similar to the shape vectors used by Skillicorn & Cai (1993) in their cost model for the Bird-Meertens formalism.

The rules on lists, *(Cons)*, *(Null)*, *(Hd)*, *(Tl)*, all have to unify one subexpression with a list type, introducing fresh type and size variables. The side condition that `hd` and `tl` can only be applied to lists with at least one element is captured in the constraint set. Thus, our type system can detect some cases of applying `hd` or `tl` to `nil`, which has size 0. The propagation of size information is also encoded via equality constraints in the constraint set.

In the *(Cond)* rule the maximum of both branches has to be used in order to obtain an upper bound of the costs of the expression. A bound on the size of the result will be added to the constraint set by applying the unification algorithm.

The *(Letrec)* case of the algorithm exhibits an extension of the format of type schemes. Since type schemes propagate generic type information from the `letrec` head into the `letrec` body, they also have to propagate the constraints collected while inferring the type of $\mathtt{e}_1$. Together with a generic type variable this constraint set is added to the result type of a `letrec` bound variable via the *(Var)* rule.

### An Example

Figure 6.8 presents a size and cost inference based on the above cost reconstruction algorithm by showing the main steps in an inference of the function `length`:

```
length = \  xs . if null xs
                   then 0
                   else 1 + length (tl xs)
```

The example inference for `length` avoids the use of intermediate variables as they would be generated in an actual implementation. Instead we directly insert the values of variables for which equality constraints are generated, e.g. $Int^1$ is used instead of $Int^x$ with $\{x = 1\}$. The inference is therefore best read from the leaves of the tree. The sized type and the cost are also directly inserted in every step, although the algorithm synthesises them in a bottom-up fashion when traversing the tree.

The most important parts in the inference are summarised as follows. The *(Null)* rule in branch 1 unifies the type of `xs` in the assumption set $\Gamma'$ , $\alpha'$, with a polymorphic list of unknown size, $List^l\ \gamma$. The *(App)* rule in branch 3 uses fresh symbolic size and cost functions, $f$ and $f'$, to describe size and cost of the result. These are attached to the type of `length` via the unification of $\alpha$ with the function type $List^{l-1}\ \gamma \overset{f,f'}{\to} \beta^z$. $\beta^z$ is a fresh size pattern, which propagates the size of the argument, `tl xs`, through

$$\dfrac{\dfrac{\dfrac{l, \gamma \text{ fresh}}{\Gamma' \ \vdash \ \texttt{xs} \ : \ List^l\ \gamma\ \$\ 0}(Var)}{\Gamma' \ \vdash \ \texttt{null xs} \ : \ Bool\ \$\ 1}(Null)}{\textcircled{1}} \qquad \dfrac{\dfrac{}{\Gamma' \ \vdash \ 0 \ : \ Int^0\ \$\ 0}(Int)}{\textcircled{2}}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma' \ \vdash \ \texttt{length} \ : \ \alpha\ \$\ 0}(Var) \qquad \dfrac{\Gamma' \ \vdash \ \texttt{xs} \ : \ List^l\ \gamma\ \$\ 0}{\Gamma' \ \vdash \ \texttt{tl xs} \ : \ List^{l-1}\ \gamma\ \$\ 1}(Tl)}{\Gamma' \ \vdash \ \texttt{length}\ (\texttt{tl xs}) \ : \ \beta^{f\ \text{sizeOf}\ (List^{l-1}\ \gamma)}\ \$\ f'\ \text{sizeOf}\ (List^{l-1}\ \gamma) + 2}(App)}{\Gamma' \ \vdash \ \texttt{1} + \texttt{length}\ (\texttt{tl xs}) \ : \ Int^{1+f\ (l-1)}\ \$\ f'\ (l-1)+3}(App)}{\textcircled{3}}$$

$$\dfrac{\dfrac{\textcircled{1} \quad \textcircled{2} \quad \textcircled{3} \qquad\qquad \Gamma' = \Gamma \cup \{\texttt{length} : \alpha, \texttt{xs} : \alpha'\} \qquad \alpha, \alpha' \text{ fresh}}{\Gamma' \ \vdash \ \texttt{if null xs ...} \ : \ Int^{1+f\ (l-1)}\ \$\ 2 + \max\ 0\ (f'\ (l-1)+4)}(Cond)}{\Gamma \cup \{\texttt{length} : \alpha\} \ \vdash \ \lambda\texttt{xs}... \ : \ List^l\ \gamma \ \xrightarrow{2+\max\ 0\ (f'\ (l-1)+4), f} \ Int^{\max\ 0\ 1+f\ (l-1)}\ \$\ 0}(Abstr)$$

**Figure 6.8** Inference for `length`

the function application. Therefore, $z$ is of the form sizeOf $(List^{l-1}\ \gamma)$. In this inference we apply the sizeOf function as early as possible to improve readability. In the inference algorithm this step would be performed at the root of the inference tree. Via the *(Cond)* rule the maximum over the costs of both branches in the conditional is constructed. In the size component of the result type the constraint set captures the fact that the size of the result is an upper bound for the size of both branches. As this information is captured via inequalities in the constraint set, the exact expression depends on the solution of the constraint set. Here we assume that it will yield a max expression. Finally, the *(Abstr)* rule builds the function type for the body of `length`, attaching the derived costs, $2 + \max\ 0\ (f'\ (l-1)+4)$, to this type. Thus, the reconstruction algorithm has exposed the recurrence for specifying the costs of computing this function with the size variables occurring in the argument types as free variables in the cost expression.

In order to obtain a closed form for the cost function of `length`, the cost expression has to be simplified and then matched against a library of recurrences. The details of these steps in general are discussed in the following two sections. In this case we have to use the fact that 0 is the neutral element for max and then use the first recurrence in Figure 6.12 to obtain the following closed form:

$$\mathit{length}_c\ l \;=\; 4 * l + 2$$

### 6.5.3 Simplifying Constraints

In order to define the simplification of constraints a normal form has to be defined on c-expressions. One natural choice would be to choose a sum-of-products representation as normal form. Assuming an ordering on size variables, an ordering on c-expressions can be defined by defining cost functions to be smaller and max expressions to be larger than cost expressions of a different shape. A simplification function would have to use rules like distributivity as well as basic arithmetic rules to bring c-expressions into normal form. This is a standard exercise in term rewriting and therefore not discussed in more detail.

From the presentation of the algebraic unification algorithm in Figure 6.4 it can be seen that many constraints will be added to the constraint set while traversing the inference tree. For a practical implementation it will therefore be important to simplify the constraint set when adding new constraints in order to avoid the generation of huge constraint sets. This also opens up the possibility to report type errors early if the check, when adding a constraint to the set, yields an error.

One important task of the simplification algorithm is to merge intermediate symbolic cost and size functions that are generated via the *(App)* rule. For symbolic cost functions this means generating one function which is the sum of all intermediate functions. In the case of symbolic size functions this means combining the upper bounds of the intermediate functions, which are present in the constraint set, into one symbolic size functions. The example in Section 6.6 discusses this aspect in more detail.

### 6.5.4 Solving Recurrence Relations

Recurrence relations are solved by matching the simplified c-expressions obtained from the cost reconstruction algorithm with a library of recurrence relations. If this match is successful the recurrence can be replaced with the known closed form. If it fails the cost expression has to be replaced with $\omega$.

This approach has the advantage of being tunable by adding more recurrence relations to the library. Thereby, it is possible to trade accuracy for speed in the analysis. Another important aspect is that the library only has to contain upper bounds rather than exact solutions to the recurrences. Thereby unsolvable recurrences can be dealt with, by choosing an approximation, possibly using a table hypergeometric functions, which is a standard technique in combinatorics. Keeping the recurrence solver separate from the derivation of the constraint set adds flexibility to the system.

A first version of a matching procedure is given in Figure 6.9. This version shows that in principle the matching procedure corresponds to a unification of c-expressions. All constants have to match exactly. C-expressions are substituted for c-variables. In compound expressions the result is the composition of all substitutions resulting from unifying the components. If this algorithm succeeds in unifying the derived c-expression with the body of a function in the library, then the resulting substitution has to be applied to the recorded closed form for this recurrence in order to eliminate the recurrence in the derived cost expression.

The correspondence of the matching algorithm to a unification algorithm also indicates that the cost for finding a closed form should not dominate the analysis. In total this cost will depend on the number of entries in the library. However, by sharing common structures in the representation of the recurrences it should be possible to devise a matching algorithm whose cost does not increase linearly with the number of entries. Refinements of fast string matching algorithms should be applicable here.

An alternative approach for eliminating recurrences at this stage would be to use a general recurrence solver over integer values. Such algorithms are available in computer algebra systems such as Maple (Char et al. 1991) and Mathematica (Wolfram 1988). This would extend the approach of using the Omega test for checking satisfiability of a constraint set to using more powerful, but more expensive, computer algebra algorithms for finding a solution for a system of recurrences. However, with the current state-of-the-art it is only possible to find closed forms for linear recur-

$$\mathcal{U}_c \;::\; c \to c \to \theta$$

$$\begin{array}{lcll}
\mathcal{U}_c\ n\ n & = & [] & \\
\mathcal{U}_c\ \omega\ \omega & = & [] & \\
\mathcal{U}_c\ l\ c & = & [c/l] & \\
\mathcal{U}_c\ c\ l & = & [c/l] & \\
\mathcal{U}_c\ (c_1 + c_2)\ (c_1' + c_2') & = & \theta'\theta & \theta = \mathcal{U}_c\ c_1\ c_1' \\
 & & & \theta' = \mathcal{U}_c\ \theta c_2\ \theta c_2' \\
\mathcal{U}_c\ (c_1 - c_2)\ (c_1' - c_2') & = & \theta'\theta & \theta = \mathcal{U}_c\ c_1\ c_1' \\
 & & & \theta' = \mathcal{U}_c\ \theta c_2\ \theta c_2' \\
\mathcal{U}_c\ (n * c)\ (n * c') & = & \mathcal{U}_c\ c\ c' & \\
\mathcal{U}_c\ (\max\ c_1\ c_2)\ (\max\ c_1'\ c_2') & = & \theta\theta' & \theta = \mathcal{U}_c\ c_1\ c_1' \\
 & & & \theta' = \mathcal{U}_c\ \theta c_2\ \theta c_2' \\
\mathcal{U}_c\ (f\ c_1\ \ldots\ c_n)\ (f'\ c_1'\ \ldots\ c_n') & = & [f/f']\theta_n \cdots \theta_1 & \theta_1 \ = \ \mathcal{U}_c\ c_1\ c_1' \\
 & & & \qquad \vdots \\
 & & & \theta_n \ = \ \mathcal{U}_c\ (\theta_1 \cdots \theta_{n-1} c_n) \\
 & & & \qquad\qquad (\theta_1 \cdots \theta_{n-1} c_n')
\end{array}$$

**Figure 6.9** Matching of cost expressions

rences with polynomial coefficients, certain non-linear first-order recurrences, and certain divide-and-conquer recurrences (Petkovšek 1990). A good starting point for such an implementation would be the algorithm by Petkovšek & Salvy (1993) for finding all hypergeometric solutions of linear recurrences. Hypergeometric sequences are such that the quotient of two successive terms is a fixed rational function of the index.

### 6.5.5 Correctness Issues

This section presents a list of conjectures characterising some crucial correctness issues of the cost reconstruction algorithm. Proving these conjectures would lead most of the way towards proving the correctness of the algorithm. No formal correctness proof will be given here. However, because the algorithm is based on the reconstruction algorithm by Reistad & Gifford (1994), and similar reconstruction algorithms for effect systems such as Talpin & Jouvelot (1992), most rules follow from their system. The main difference is the *(App)* rule, which propagates size information.

In order to formalise the notion of solving a set of constraints the notion of a model has to be introduced. In this definition we assume the standard definitions of $=$ and $\leq$ on integer values with $x \leq \omega$ for all integer values $x$.

**Definition 7 (solution, model)** *A mapping $\psi$ from c-variables to c-expressions is a solution of a constraint set $C$ (written $\psi \models C$) iff for all elements $c_1\ R\ c_2$ of $C$, $\psi c_1\ R\ \psi c_2$ where $R \in \{=, \leq\}$.*

Based on this definition several conjectures over models for composed constraint sets such as the following can be formalised.

**Conjecture 2** *Let $C_1, C_2$ be constraint sets. Then*

$$\psi \models C_1 \cup C_2 \quad \text{implies} \quad \psi \models C_1 \quad \text{and} \quad \psi \models C_2$$

This formalises the intuition that a solution of a composed constraint set must be a solution of every component.

The soundness of the cost reconstruction algorithm with respect to the inference system can be formalised as follows. The symbol $\vdash$ is used to represent the sized time inference system in Figure 6.1 and $\vdash_{alg}$ is used to represent the cost reconstruction algorithm in Figures 6.5 and 6.6.

**Conjecture 3 (soundness of cost reconstruction)** *Let $\mathtt{e}$ be a $\mathcal{L}$ expression and $\Gamma$ the initial assumption set for type inference. Then*

$$\Gamma \vdash_{alg} \mathtt{e} : \langle \tau, \theta, c, C \rangle \quad \text{and} \quad \psi \models C \quad \text{implies} \quad \psi\theta\Gamma \vdash \mathtt{e} : \psi\theta\tau\ \$\ \psi\theta c'$$

*where $\psi\theta c' \leq \psi\theta c$.*

The inequality in this conjecture is caused by the possibility of deriving finite upper bounds for recursive functions if $\psi$ contains finite solutions to the symbolic cost functions in $C$. If $\psi$ maps all symbolic cost functions to $\omega$ or if $\mathtt{e}$ is a non-recursive expression this inequality can be tightened to an equality. Currently, this is not expressed in the sized time system.

The structure of the proof has to be as follows. The proof performs a case analysis over the $\mathcal{L}$ expressions and, in each branch of the case analysis, a structural induction of the inference/algorithm tree. The induction assumption is the soundness conjecture

```
pay_price = \ price coins .
 if (price==0) then 1
 else
  letrec  coin_values = nub (dropWhile (\ x . x>price) coins)
  in
   par coin_values
       (sum (map (choose price coins) coin_values))

choose = \ price coins c .
  letrec   new_coins' = dropWhile (\ x . x>c) coins
           new_coins  = del new_coins' c
  in
   par new_coins
       (pay_price (price-c) new_coins)
```

**Figure 6.10** $\mathcal{L}$ code for `coins`

for all subexpressions. It uses several conjectures on substitutions and models stated above. The structure of the proof is similar to a proof for the soundness of effect system as given in Talpin & Jouvelot (1992). The main difference to this system is the use of algebraic unification in the system presented here.

## 6.6 Example

This section gives an abridged size and cost inference, performed by hand, for one function in the simple but non-trivial $\mathcal{L}$ program, `coins`. The inference shows how to infer size and cost information by using the sized time system. Since the goal is to generate information that can be used in the runtime-system to improve performance, this inference focuses on the size and cost bounds that can be derived without giving all details of the inference process and how the constraints are collected. Finally, this section concludes with giving performance measurements of the program annotated with the derived cost information.

The `coins` program takes a price and a list representing a set of coins, and determines how many different combinations of coins could be used to pay for an object at the given price. The full code of the program is given in Figure 6.10. The code exposes parallelism via the `par` annotation. The goal of the granularity analysis is to infer cost and size expression, which can then be added to the `par` annotations. Figure 6.13 will present the annotated version of the code that makes use of the derived information.

$$\dfrac{}{\Gamma'' \vdash \texttt{del} \ : \ \alpha \ \$ \ 0}(Var) \qquad \dfrac{}{\Gamma' \vdash \texttt{zs} \ : \ List^{l-1} \ \gamma \ \$ \ 0}(Var)$$

$$\dfrac{}{\Gamma'' \vdash \texttt{del zs} \ : \ {\beta_1}^{f_z \ \mathrm{sizeOf} \ (List^{l-1} \ \gamma)} \ \$ \ 1 + f_c \ \mathrm{sizeOf} \ (List^{l-1} \ \gamma)}(App) \qquad \dfrac{}{\Gamma'' \vdash \texttt{x} \ : \ \gamma \ \$ \ 0}(Var)$$

$$\dfrac{\dfrac{\dfrac{}{\Gamma'' \vdash \texttt{del zs x} \ : \ {\beta_2}^{z'} \ \$ \ 2 + f_c \ (l-1) + f_c'}(App)}{\Gamma'' \vdash \texttt{cons z (del zs x)} \ : \ List^{1+z'} \ \gamma \ \$ \ 2 + f_c \ (l-1) + f_c'}(Cons)}{\Gamma'' \vdash \texttt{if ...} \ : \ List^{1+z'} \ \gamma \ \$ \ 2 + \max \ 0 \ (2 + f_c \ (l-1) + f_c')}(Cond)$$

4 $\quad \Gamma'' = \{\texttt{z} : \gamma, \texttt{zs} : List^{l-1} \ \gamma, \texttt{xs} : List^{l} \ \gamma, \texttt{del} : \alpha\}$

**Figure 6.11** A part of the inference of `del`

In order to focus the presentation, this section will only discuss the inference of one sub-function, `del`. This function takes a list and a value and deletes the value from the list. If the value is not in the list an error value is returned. The definition of `del` in $\mathcal{L}$ is as follows.

```
del =  \ xs x . if (null xs) then error
                else letrec  z = hd xs
                             zs = tl xs
                     in
                      if (z==x) then zs else cons z (del zs x)
```

The `del` function deletes the first instance of `x` from `xs`. The special value `error` (of the polymorphic type $\forall\beta.\beta^{-1}$) is used to indicate that `x` did not occur in `xs` (an error). Its size has to be smaller than any list size.

### 6.6.1 Cost and Size Analysis

This section highlights the main points in each of the steps for performing the cost and size inference as presented in Section 6.5.1. This example will also discuss limits of this inference and requirements for the simplification algorithm that has to be used after performing cost and size reconstruction.

**Inference and Simplification.** In this step the inference tree is generated and traversed. During the tree traversal constraints on c-expressions are collected. The most important part in this traversal is the inference of the inner conditional. This part of the inference tree is shown as branch 4 in Figure 6.11. Before reaching branch 4 the analysis of `null xs` in the head of the outer conditional adds the following binding to the assumption set: `xs` : $List^l\ \gamma$. Furthermore, in the analysis of the `letrec` construct `zs` is defined to be `tl xs` and therefore its type is $List^{l-1}\ \gamma$. Similarly, the type of `z` is $\gamma$. In branch 4 the head of the inner conditional unifies the type $\gamma$ with the type of `x`. In the two application rules the size information of the concrete arguments is added to the size pattern representing the result type. The result of applying the sizeOf function represents the change in size for the arguments of the recursive function call:

$$\begin{aligned}&\text{sizeOf}(List^{l-1}\ \gamma) = [l-1]\\&\text{sizeOf}(\gamma) = []\end{aligned}$$

The information from the first application is propagated to the second application via unifying the size pattern with a function type containing fresh symbolic size and cost functions, $f'_z$, $f'_c$:

$$\mathcal{U}({\beta_1}^{f_z\ \text{sizeOf}\ (List^{l-1}\ \gamma)}, \gamma \overset{f'_c, f'_z}{\rightarrow} {\beta_2}^{f'_z\ \text{sizeOf}\ \gamma})$$

In this case, the unification algorithm will choose an upper bound for the size in both size patterns, $z'$, and use it as the size of the result. The constraint set will therefore contain the following inequalities

$$\{f_z\ (l-1) \leq z', f'_z\ \leq z'\}$$

and the result type of `del zs x` is the size pattern ${\beta_2}^{z'}$. This example shows that the reconstruction algorithm adds fresh size functions in each curried function application. In order to obtain just one size function over all size arguments these functions have to be merged. This should be done by the simplification algorithm when generating a size function from the size expression for the body of the function definition. The same merging has to be done for cost functions. However, in this case all symbolic cost functions, $f_c$ and $f'_c$ in this case, occur explicitly in the resulting sum.

The *(Cons)* rule unifies the size pattern to the list type $List^{1+z'}$. Then the *(Cond)* rule constructs the maximum of the size of both branches, with $-1$ as the size for the `then` branch. Collecting all size and cost information in the inference tree the following two recurrences are exposed on top-level:

$$\begin{aligned} del_{\mathcal{Z}}\ l &= \max\ (-1)\ (1 + del_{\mathcal{Z}}\ (l-1)) \\ del_{\mathcal{C}}\ l &= 2 + \max\ 0\ (4 + \max\ 0\ (2 + del_{\mathcal{C}}\ (l-1))) \end{aligned}$$

In these cost expressions the occurrences of max reflect the two nested conditionals in the code. In this case the base case for the recurrence can be obtained by reinterpreting the max operator as a minimum and choosing the minimum size as argument. In the general case, however, this would require a more sophisticated analysis of the head of conditionals. In particular, the semantics of `null` should be used together with the available size information. One promising approach to achieve this would be the use of conditional types as outlined in Section 6.5.1.

After simplification, this stage of the inference yields the following recurrences:

$$\begin{aligned} del_{\mathcal{Z}}\ 0 &= -1 \\ del_{\mathcal{Z}}\ l &= 1 + del_{\mathcal{Z}}\ (l-1) \\ \\ del_{\mathcal{C}}\ 0 &= 2 \\ del_{\mathcal{C}}\ l &= 8 + del_{\mathcal{C}}\ (l-1) \end{aligned}$$

**Resolving Recurrences.** The goal of this step is to bring all symbolic cost functions (like $del_{\mathcal{C}}$) into closed form in order to substitute the functions with the expressions in the constraint set. This will eliminate all symbolic cost functions introduced by the reconstruction algorithm. By using a library of known recurrences the recursive size and cost functions above can be replaced by the following closed forms:

$$\begin{aligned} del_{\mathcal{Z}}\ l &= l - 1 \\ del_{\mathcal{C}}\ l &= 8 * l + 2 \end{aligned}$$

It is important to note that all recurrences in the analysis of these functions are linear, first-order recurrences since the functions iterate over lists. Figure 6.12 shows the entire library of closed forms for recurrences that has been used in the analysis of `coins` and its subfunctions.

**Solving the Constraint Set.** The final step has to check whether a solution for the constraint set exists. In this case the program is well typed and for each function

$$
\begin{array}{lcl}
f\ 0 & = & a \\
f\ n & = & b + f(n-1)
\end{array}
\Rightarrow\ f\ n\ = a + b * n
$$

$$
\begin{array}{lcl}
f\ 0 & = & a \\
f\ n & = & b + c * n + f(n-1)
\end{array}
\Rightarrow\ f\ n\ =\ a + b * n + \frac{c*n*(n+1)}{2}
$$

**Figure 6.12** Recurrences and their closed forms

a corresponding size and cost function has been inferred. Since the constraint set does not contain symbolic cost functions any more at this stage the Omega test can be used for performing this check. An analysis of the expressions in `coins` that are annotated with `par` and should be evaluated in parallel (`coin_values` and `new_coins`) yields the following cost expressions that are used to annotate the program:

$$
\begin{array}{lcl}
coin\_values_{\mathcal{C}} & = & 9 * n^2 + 14 * n + 5 \\
new\_coins_{\mathcal{C}} & = & 16 * n + 4
\end{array}
$$

### 6.6.2 Annotations

The cost information derived in the previous section can be used to transform the parallel program by adding cost information to the spark sites:

1. For each argument add an extra argument representing its size.
2. Use the derived size functions to propagate size information.
3. Add the derived cost expressions to the `parGlobal` annotations.

This transformation applied to the input program shown in Figure 6.10 gives the annotated parallel program shown in Figure 6.13. Note that the new variables `m` and `n` represent the size of `price` and of `coins`, respectively.

The first argument of `parGlobal` represents a cost or granularity measure. The `parmap` function is a parallel implementation of `map` that takes granularity information for each application of the mapped function as its first argument. The special value `infty` represents $\omega$ as a bound on computation cost.

```
pay_price = \ m n price coins .
 if (price==0) then 1
 else
  letrec  coin_values = nub (dropWhile (\ x . x>price) coins)
  in
   parGlobal (9*n^2+14*n+5) coin_values
             (sum (parmap infty (choose m n price coins) coin_values))

choose = \ m n price coins c .
  letrec   new_coins' = dropWhile (\ x . x>c) coins
           new_coins  = del new_coins' c
  in
   parGlobal (16*n+4) new_coins
             (pay_price m (n-1) (price-c) new_coins)
```

**Figure 6.13** Annotated $\mathcal{L}$ code for `coins`

### 6.6.3 Measurements

This section presents results on running the annotated program under the GranSim simulator in two different set-ups: with eager-thread-creation and with evaluate-and-die. Figure 6.14 compares the granularities over varying cut-off values when using a thresholding granularity improvement mechanism. The cut-off value is measured as recursion depth starting with 100 at the root of the divide-and-conquer tree. In both cases the results for several different latencies are plotted. In the case of eager-thread-creation (left hand graph) the granularity increases gradually with increasing cut-off values. In the case of high latency (4,096 cycles) the granularity turns out to be rather good already. The graph on the right hand side shows a similar, continuous improvement of the granularity. Only at a few points a reduction of granularity is observed. This corresponds to the mismatch between upper bounds of computation costs and the real costs.

The improvements in speed-up are smaller but still significant. In the case of eager-thread-creation the speed-up increases from 14.3 to 18.4 for a latency of 64 cycles. For higher latencies the improvement is smaller but still measurable. In the case of an evaluate-and-die model, however, only very small improvements can be observed. The right hand graph in Figure 6.14 already shows a high granularity for low cut-off values. Only for a latency of 4,096 cycles there is clear improvement in speed-up from 24.7 to 26.1. The main reason for this behaviour is spark subsumption in the evaluate-and-die model. Whereas eager-thread-creation without thresholding creates more than 10,000

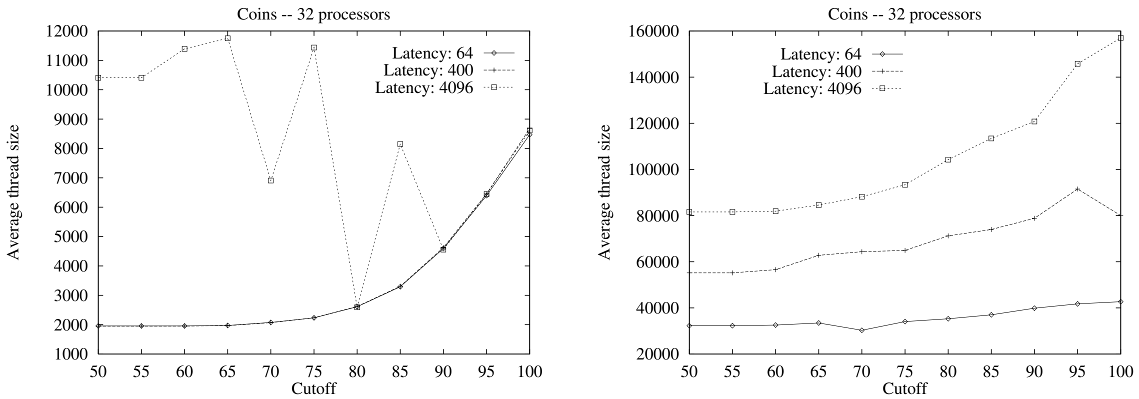


**Figure 6.14** Granularity with varying cut-off values (eager and lazy thread creation)

threads, the evaluate-and-die model only creates circa 1,000 threads. These results of granularity being important in particular for high-latency machines, correspond to the results of measurements on the distributed memory Alfalfa architecture reported by Goldberg (1988*a*).

In evaluating the performance improvement by adding granularity information it has to be emphasised that this program contains only two main spark sites. This severely limits the amount of runtime improvement that can be expected by adding granularity information. Granularity control mechanisms mainly aim at improving programs with a large number of spark sites generating tasks whose granularities vary significantly (see Chapter 5). This is, for example, the case for naive methods of generating implicit parallelism in a functional program. Another result of these measurements is the observation that it is possible to achieve runtime improvements for a wide range of latencies representing different kinds of parallel architectures.

## 6.7 Comparison with Other Work

### 6.7.1 Complexity Analysis

Pioneering work on automatic complexity analysis has been done by Wegbreit in developing a system METRIC for deriving closed form expressions for the time complexity of a first-order subset of Lisp (Wegbreit 1975). The structure of his analysis is somewhat similar to the proposed cost inference algorithm discussed in this thesis (see Section 6.5):

1. Local cost assignment translating a program into a set of cost expressions.
2. Recursion analysis determining how the parameters to a recursive function change from one call to another.
3. Solution of difference equations using standard methods like direct summation and differentiation of generating functions.

In his concluding remarks Wegbreit points out how a sophisticated algebraic manipulation subsystem and an enhanced difference equation solver could dramatically improve the quality of the results produced by the system. This would be equally true for the granularity analysis of functional languages using a general recurrence solver because the differences to imperative languages treated by Wegbreit only complicates the generation but not the manipulation of cost expressions.

Wegbreit's work has been extended by Hickey & Cohen (1988), who focus on theoretical foundations of a performance compiler capable of automatically generating functions describing average-case performance. The systems Complexa (Zimmermann 1990) and $\Lambda\Upsilon\Omega$ (Flajolet et al. 1991) build on METRIC and extend it for the average-case complexity analysis of algorithms. Skillicorn & Cai (1993) as well as Rangaswami (1996) use cost models based on the Bird-Meertens calculus in order to obtain information about the runtime of parallel programs. In a similar spirit Jay et al. (1997) develop and implement a monadic cost calculus for a higher-order functional language. This language is restricted in away that makes it possible to derive the shape of the result of an expression based on the shape of its inputs. Thus, shape information is available for all program expressions. This corresponds to our use of sized types but shape information is more accurate because the size annotations in the type system for $\mathcal{L}$ are only upper bounds. In Jay et al. (1997) programmer estimates on the number of unfoldings for recursive functions are required to obtain accurate costs. It is demonstrated that with this information the implemented calculus automatically derives parallel execution times for programs like matrix multiplication. One technical difference to the monadic cost calculus is that the sized time system, inspired by effect systems, uses an extended type system to propagate information about sizes and costs, whereas Jay et al. (1997) use a monad for this purpose. The close relationship between effect systems and monads has been recently elaborated by Wadler (1998).

### 6.7.2 Cost Analysis for Strict Languages

Huelsbergen et al. (1994) introduce the technique of a *dynamic granularity estimation* for strict, list-based, higher-order languages. This technique consists of two components:

- A compile-time (static) component, based on abstract interpretation to identify components whose complexity depends on the size of a data structure.
- A run-time (dynamic) component, for approximating sizes of the data structures at run-time.

Based on the results of the static component, the compiler inserts code for checking the size of parameters at certain points. At runtime the result of these checks determine whether a parallel task is created or not. The static component of this system has not been implemented. The dynamic component is implemented on a Sequent Symmetry on top of a parallel SML/NJ implementation. It is stated that the runtime overhead for keeping track of approximations (one additional word per cons cell) is very low. For the quicksort example an efficiency improvement of 23% has been reported.

Dornic (1993) describes a practical *time system* for inferring a function's complexity using an algorithm similar to the cost reconstruction algorithm presented in this thesis. In Dornic's time system, however, recursive functions are assigned infinite costs as an upper bound for the total computation time. In part based on Dornic's work, Reistad & Gifford (1994) define the notion of *static dependent costs* for the analysis of a strict higher-order language. These costs describe the execution time of a function in terms of its input. The relationship of our sized time system to to the work on static dependent costs has been discussed in detail in Sections 6.4.1 and 6.5.2. Runtime measurements of the system show that their cost estimates are usually within a factor of two of the real costs. Using this information for a parallel map operation achieved a speedup of more than two compared to a naive version of a parallel map on a four processor SGI for the game of life program.

Rosendahl (1989) deals with a complexity analysis of a first-order subset of Lisp. His work builds on a partial evaluation machinery and uses abstract interpretation in order to derive upper bounds for the complexity of first-order Lisp functions. The analysis has three phases: constructing a step-counting version of the given program; perform abstract interpretation on the step-counting version of the program and generate a time bound function; finally, simplify the resulting time bound function. The latter includes a component for solving finite-difference equations (Rosendahl 1986), which is similar to our approach of using a library of recurrences.

In all of the above analyses user defined recursive functions are assigned infinite costs, except for Rosendahl (1989) where simple recursive patterns can be eliminated. More recently, Hughes et al. (1996) have developed a sized type system for a simple higher-order, lazy functional language. This type system allows to infer upper bounds for the size of algebraic data types. In the mentioned paper this is used to prove termination and liveness of reactive system. However, this thesis demonstrates that a sized type system can also be used to analyse the costs of user-defined recursive functions.

The ACE system of Le Métayer (1988) transforms an FP program with call-by-name semantics into a program with call-by-value semantics. The main part of the system is the transformation of recursive complexity functions into non-recursive ones. In contrast to the approaches mentioned above, this system performs a macro-analysis, that is, it measures the time in the number of applications of the dominant operation which is used in the program.

### 6.7.3 Demand Analysis

The purpose of a demand analysis is to determine the order in which parts of an expression are needed and the degree to which the result has to be evaluated. In a lazy language this information is required to determine the computation costs of an expression. Demand analysis is similar to strictness analysis but it provides more detailed information. In fact, strictness information can be extracted as a special case from the information provided by demand analysis.

In the context of a cost analysis it is important to know the order-of-demand as well as the degree of the evaluation of a complex data structure. Several approaches have been proposed to perform both kinds of analysis:

**Abstractions of sets of continuations:** Both Hughes (1987) and Bloss (1989) define a collecting non-standard semantics of all possible continuations (or paths) in a program. An order-of-evaluation analysis is developed by defining an abstract interpretation over this semantics. The main disadvantage of this approach is that the resulting terms in the abstract interpretation are very big, and hence algebraic simplification is necessary to derive normal forms for these terms. Exact simplification is not always possible and heuristics have to be applied at certain points. In contrast, the inference based analysis presented in Draghicescu & Purushothaman (1990) does not try to enumerate all possible paths and is more efficient in practice. Similar order-of-demand analyses have been developed by Park & Goldberg (1992) and Gomard & Sestoft (1991).

**Many-valued evaluation degrees:** In the framework of Martin-Löf type theory Bjerner (1989) develops many-valued evaluation degrees, which are used to give an operational model of contexts. This approach usually gives very accurate results but it is less general than the projections approach. Many-valued evaluation degrees have been developed specifically for time analysis. Because they do not contain more information than absolutely necessary for a time analysis symbolic derivations are easier.

**Projections:** The property $\pi \sqsubseteq ID$ of a projection $\pi$ can be read as $\pi$ performs an evaluation of a degree less than or equal to that of reduction to normal form. Based on this observations Wadler & Hughes (1987) developed a strictness analysis, which uses projections to model demand. Projection transformers are used to determine the demand on an argument in a function application, given the demand on the whole function application. This corresponds to the way that evaluation transformers (Burn 1991*a*) determine the degree of evaluation in a parallel environment. The compilation rules developed by Burn (1990) show how the information provided by projections can be exploited in both sequential and parallel implementations.

The most promising approach is the use of projections, which have recently attracted a lot of attention for static analysis in general (Davis 1994). This is underlined by recent work on the theoretical foundations of projections (Launchbury & Baraki 1996) as well as the use of projections in the implementation of a strictness analyser in the Glasgow Haskell Compiler (Kubiak et al. 1991). An implementation of a demand analysis could reuse a lot of this work. However, it is an open question whether the concrete set of projections used in this implementation is strong enough to allow satisfactory cost information to be inferred.

### 6.7.4 Cost Analysis of Lazy Languages

Based on the modelling of demand via many-valued evaluation degrees in Bjerner's PhD thesis (Bjerner 1989), Bjerner & Holmström (1989) develop a cost analysis for lazy higher-order languages. A separate demand analysis is used to derive information on the evaluation degree.

In his PhD thesis Sands (1990*a*) uses projections in order to develop a cost calculus for a lazy, higher-order language. He specifies cost calculi for inferring a lower bound, necessary time, and an upper bound, sufficient time, of the cost for evaluating an expression. This work is partly based on Wadler's use of projections for the time analysis of lazy programs (Wadler 1988). Being calculi rather than static analyses both approaches assume knowledge about exact values e.g. in the head of conditionals. To date Sands' cost calculus seems to be the most promising basis for a concrete implementation of a cost analysis for lazy languages.

### 6.7.5 Logic Languages

In the area of logic programming languages some attempts have been made to combine a cost analysis (Debray et al. 1990, Debray et al. 1994, Tick & Zhong 1993) with runtime mechanisms for improving the granularity of the generated threads (López García et al. 1994, López García et al. 1995). The cost analysis of logic languages is complicated by the fact that a relation can have several solutions. Thus, a separate number-of-solutions analysis has to be developed to infer this information (Debray & Lin 1993). The structure of the program transformations using cost information is similar to those in functional languages: add the cost functions derived at compile time to the code; generate a parallel as well as sequential version of the code; add conditionals for deciding whether to use the sequential or the parallel code. Several optimisations to minimise the runtime overhead of these methods have been developed. The most important optimisation is to simplify the size expressions that are generated (Hermenegildo & López García 1995). However, some overhead is inherent in such a hybrid approach and there is still the danger of code explosion.

## 6.8 Discussion

This chapter has shown how to infer upper bounds for the size of the result and the computation cost of evaluating an expression in the simple strict higher-order functional language $\mathcal{L}$. The sized time system has not yet been implemented. However, based on the results by Hughes et al. (1996) the implementation of a time checking algorithm should be a straightforward extension of their sized type checking algorithm. In order to extend this algorithm to time inference, the analysis has to solve recurrence equations over an integer domain. The cost reconstruction algorithm in Section 6.5.2 shows how to expose recurrences for recursive functions. These recurrences can then be solved by matching them with a library of known recurrence relations and (an approximation of) their closed forms. An algorithm for combining the cost reconstruction algorithm with such a library has been outlined and open problems have been discussed. The library approach makes it possible to derive costs for many user-defined recursive functions, which goes beyond the analysis presented by Reistad & Gifford (1994) for Lisp. A similar approach by Rosendahl (1986) shows that many common patterns of computation can be analysed with a rather small set of recurrences. In the context of parallel computation it is important to obtain exact information for small functions that usually generate simple recurrences. Therefore, a small library should be sufficient to yield useful information.

Several stages in the inference algorithm outlined in Section 6.5.1 need refinement in order to implement the full algorithm. In particular, the simplification algorithm has to merge symbolic cost and size functions, and determining the costs for the base case of a recursion requires in general a more sophisticated analysis. These issues will be discussed further in the context of future work in Section 7.3.

Although the derived cost is only an upper bound for the real cost, the initial measurements indicate that it can provide enough information for the runtime-system to achieve a performance improvement of parallel programs. This is quite remarkable because the analysis was performed for a strict language and is therefore overestimating the evaluation degree in the presented measurements. This seems to give evidence that at least for strict functions in a lazy language the results of a strict analysis, such as the sized time system, can provide useful information. However, before making conclusions on this issue more measurements of analysed programs, especially large-scale programs, are required.

From the measurements in Chapter 5 it is unclear whether relative cost information between threads is sufficient to achieve performance improvements. Therefore, the presented analysis yields absolute cost information. In the measurements for a hand analysed program the use of absolute cost information via a thresholding mechanism achieved the best results. The accuracy of the analysis could be improved, however, by adding constants for certain operations rather than performing step counting alone.

The presented analysis is based on type inference rather than abstract interpretation, which is often used for this kind of static analysis. The main advantages offered by an inference-based analysis are its modularity, by propagating all relevant information via the type of an expression, and its tunability, in particular when using a library approach in order to eliminate recurrences. Both issues are particularly important for the analysis of large programs. Therefore, the algorithm outlined in this chapter should be a good basis for a practical implementation.

# *Chapter 7*

# Conclusions

## 7.1 Summary

To develop a system of implicit parallelism for lazy functional languages a sophisticated runtime-system has to be built. It must achieve good parallel performance without a detailed description of the parallel program execution from the programmer. It must be flexible enough to deal with programs of very different structure, but should also be able to make use of certain important characteristics of the program. This thesis focuses on one of these characteristics, the granularity of the generated threads in a parallel system, and it furthers this effort by developing and measuring *granularity improvement mechanisms* for the runtime-system, and developing a *static granularity analysis*, based on sized types (Hughes et al. 1996) and a time system (Reistad & Gifford 1994), for inferring an upper bound of the computational costs of evaluating a program expression. This thesis also contributes to the development of a systematic programming technique for parallel lazy functional programming, *evaluation strategies*, which achieves a clean separation between algorithmic and behavioural code. The main contribution of this thesis to this part, strategic function application, has proven useful for several large application programs, in particular in the top-level parallelisation of Lolita. The programming style used in the parallelisation, *data-oriented parallelism*, makes use of laziness in order to specify the parallelism over a data structure independently from its definition and thus facilitates a top-level approach towards parallelisation, in which the parallelism is specified at the top-level without having to change individual components of the program.

One of the fundamental questions addressed by this thesis is: can the parallel performance of functional programs with sequential lazy evaluation and a parallel evaluate-and-die model of computation be improved when adding granularity information? From the discussion in Chapter 5 this seems to be true for a class of parallel programs where the granularity of generated sparks does not monotonically decrease during the program execution. For simple divide-and-conquer examples this monotonicity means that the FIFO management of the spark pool is sufficient to achieve good granularity in practice. However, for unbalanced divide-and-conquer problems an explicit thresholding mechanism or a priority based management of the spark pool can achieve better performance. In the experiments presented here it is shown that the elimination of small threads via a simple thresholding mechanism achieves the biggest improvement of about a factor of two in speedup.

The presented granularity improvement mechanisms should also be useful to improve the parallel behaviour on massively parallel systems with thousands of processors. In these systems it is unlikely that an evaluate-and-die mechanism can subsume many sparks, because the ratio of generated sparks to runnable threads will be much smaller. This increases the probability of a spark being picked up by an idle processor before its work is subsumed by another thread. But it would still be advantageous to eliminate tiny threads whose creation cost is higher than their total computation. We have not been able to investigate this aspect of scalability, however, because the system-oriented view of GranSim is currently limited to 64 processors.

As a test platform for the granularity improvement mechanisms GranSim has been developed. GranSim is a highly parameterised and accurate simulator for the parallel execution of GpH programs. It combines lazy evaluation with an evaluate-and-die model of parallelism. It is integrated into a state-of-the-art compiler forming an important component of an engineering environment for parallel program development. It provides setups for both idealised simulation and realistic simulation with a detailed modelling of communication. GranSim is also highly parameterised to model a variety of parallel machine architectures and this has proven very important for the performance tuning of parallel programs.

The combination of all these features makes GranSim unique. Most existing simulators only count reduction steps rather than machine instructions executed by optimised compiled code. Also the parameterised modelling of communication costs is unusual for simulators. With the availability of all GHC optimisations it is possible to investigate the influence of the latest sequential optimisations on the parallelism in the program.

A complementary step in devising a system that makes automatic use of granularity information is to derive this information and to make it available to the runtime-system. Several methods to do this have been suggested in the literature: profiling approaches, ad-hoc heuristics etc. In this thesis a static analysis is used in order to minimise the overhead for the runtime-system. The granularity analysis that has been presented in Chapter 6 builds on top of existing analyses and derives an upper bound for the computation costs measured as abstract computation steps. As a refinement of the analysis it would be possible to model concrete costs of the computation model and of the parallel machine via constants that can be added to the analysis. Although not all parts of the inference have been rigorously specified, a detailed outline of the inference algorithm has been given. The experimental results with hand-analysed programs show that this can provide useful information for the runtime-system.

One of the main limitations of a static analysis for extracting granularity information is its inability to make use of concrete runtime data. In particular it is not possible to make some kind of branch prediction for conditional constructs. However, the presented analysis could be extended in several ways in order to alleviate this problem. One possibility would be to extend the type system further to capture runtime information via conditional types. Thereby, the type would encode the relationship between the head of the conditional and the branches. An alternative would be to rely on profiling data in order to obtain information on the probabilities of the branches. This information could then be used as weights for the costs of the branches. Finally, the granularity analysis could be augmented with a separate analysis that tries to extract boolean values out of program expressions, using the available size information. For example such an analysis could determine the value of calls to the `null` function, which only needs information on the size of the list. If exact size information is available at compile time, the computation path through conditionals depending on `null` could be predicted.

In parallelising a set of large functional programs a purely annotation based approach proved to be not entirely satisfactory. This has led to the development of evaluation strategies, in a group effort, and of strategic function application, in particular. In order to describe the dynamic behaviour of a function call, strategic function application parameterises normal function application with a strategy specifying evaluation degree and parallelism. The resulting data-oriented style of programming achieves a modularity of program components and a separation between algorithmic and behavioural code not usually found in strict languages. This is mainly due to the decoupling of the data structure's generation and the specification of its parallelism, which helps to maintain the abstraction provided by modules and functions. In contrast, strict languages tie the evaluation of an expression to the point of its definition. Therefore, it is much harder to separate the definition of a value from the parallelism in computing this value. A comparison of several versions of a parallel linear system solver, LinSolv, has demonstrated that a data-oriented parallel programming style is superior to the naive use of parallelism combinators. The use of evaluation strategies in the parallelisation of programs as large as Lolita showed that the additional code for parallelisation can be localised to a high degree, in this case to only two out of circa three hundred modules.

Studying large, lazy, parallel programs is rarely done but in creating a powerful engineering environment for parallel programming it is important in order to evaluate:

1. The suitability of evaluation strategies to realistic functional programs. While working on the parallelisation of Lolita the repetition of some clumsy constructs in an initial version was the main motivation for introducing strategic function application.

2. The impact of laziness on parallel programming. Laziness favours a top-down approach for parallelisation, in particular data-oriented parallelism. This aspect is demonstrated in the parallelisation of Lolita in Section 4.5. However, although it is easy to add parallelism it is often hard to predict the effects for complex parallel programs.

3. The completeness of the existing set of visualisation tools for performance tuning i.e. whether the tools provide sufficient information to the programmer for tuning the performance of a parallel program. The importance of the visualisation tools has been shown in the discussion of LinSolv in Section 4.6.

## 7.2 Contributions

This section discusses the contributions of the thesis in more detail and points out research that has been undertaken jointly with other researchers. The concrete contributions of this thesis are as follows.

1. *Parallelisation of large lazy functional programs* (Loidl & Trinder 1997): In the parallelisation of several large functional programs this thesis has combined the advantages of lazy and of parallel evaluation, achieving a modular parallel programming style. A set of large algorithms has been parallelised and their performance has been tuned. These programs typify application areas such as symbolic computation and artificial intelligence. In particular, this thesis has developed a parallel imperative, a parallel pre-strategy, and a parallel strategic version of LinSolv (see Section 4.6). A comparison of both functional versions showed that the performance tuning process is significantly simplified by using strategies. This is supported by several other medium-sized strategy programs like a parallel Alpha-Beta search algorithm. The latter program demonstrates, for the first time, how strategies can express complex dynamic behaviour in programs that crucially rely on laziness. The parallelisation of Lolita in Section 4.5, the largest existing parallel non-strict functional program, showed the advantages of data-oriented parallelisation for large systems in order to parallelise code without breaking the abstraction of modules. The parallelisation of Lolita has been done in cooperation with the Computer Science Department at the University of Durham.

2. *Highly parameterised, accurate simulator* (GranSim) (Hammond et al. 1995): The GranSim simulator (see Chapter 3), which has been developed in joint work by the author in this thesis, provides, unlike most other simulators, both an idealised and an accurate modelling of a parallel machine. It is highly parameterised in order to model a wide range of parallel architectures. In using GranSim on large programs, such as Lolita, it has proven to be robust and an essential component in the parallel engineering environment built on top of the Glasgow Haskell Compiler (GHC). It closely models the features of GUM, the portable runtime-system for Haskell, which is also part of the parallel engineering environment. GranSim is publicly available and currently being used at other universities worldwide for both program parallelisation and prototyping of runtime-system features.

   The original prototype, which has been designed and implemented in cooperation with Dr. Kevin Hammond and Dr. Andrew Partridge, provided the core functionality of simulating a distributed heap, maintaining thread and spark pools, and instrumenting the code generated by GHC. The setup in this prototype used synchronous communication and single closure fetching. The major enhancements performed independently include the implementation of the idealised GranSim-Light setup, the design and several extensions of the communication system including the implementation of several variants of asynchronous communication, and of packing graph structures (Loidl & Hammond 1996*b*). The latter is based on the author's implementation of bulk fetching in GRAPH for PVM (Loidl & Hammond 1994). A large set of visualisation tools, showing activity and granularity at several levels of detail, has been developed for GranSim. Furthermore, GranSim has been integrated into GHC and is now available for both Haskell 1.2 and 1.4.

3. *Use and refinement of evaluation strategies* (Trinder et al. 1998): The author's implementation of several lazy parallel algorithms in part motivated and guided the initial design of evaluation strategies. Recoding the LinSolv algorithm using strategies contributed to the refinement of strategies. Experience with programs such as Lolita was very important for making basic design decisions. The parallelisation of several medium-sized programs produced strategies that have proven to be of general use. This thesis in particular contributed to evaluation strategies by adding strategic function application (see Section 4.3.7) to the initial version of strategies. Strategic function application parameterises function application with a strategy describing the parallelism and the evaluation degree on the function argument. The resulting programming style, data-oriented parallelism, for the first time combines the main advantages of lazy evaluation, in particular modularity, and parallel computation, reduced runtime, on a large scale. Evaluation strategies have been developed in a group effort with Dr. Phil Trinder, Dr. Kevin Hammond and Prof. Simon Peyton Jones.

4. *Static granularity analysis* (Loidl & Hammond 1996*a*): The thesis presented a static analysis for inferring upper bounds of computation costs of program expressions in a simple strict functional language (see Chapter 6). This work is based on sized types (Hughes et al. 1996) and a time system for a Lisp-like language (Reistad & Gifford 1994). However, the analysis makes it possible to handle some user defined recursive functions by exposing recurrences in the cost reconstruction algorithm and then matching these functions with a library of recurrences and their known closed forms. Although this analysis has not been

implemented, a detailed outline of a possible implementation, in particular of a cost reconstruction algorithm, is given.

5. *Implementation and measurement of runtime-system features to improve parallel performance* (Loidl & Hammond 1995): This thesis discussed several granularity improvement mechanisms that have been implemented and measured in the context of both an evaluate-and-die and an eager-thread-creation model of parallelism (see Section 5.5): priority sparking, priority scheduling and an explicit threshold mechanism. All mechanisms make use of granularity information in the source code via program annotations. They have been measured for several hand-annotated programs. As a result moderate improvements in performance have been observed especially when eliminating all small threads with a threshold mechanism.

## 7.3 Further work

### Strategies

The results of using evaluation strategies in the parallelisation of several lazy programs have been very encouraging. It would be interesting to use the same technique for the parallelisation of strict programs. We hope to achieve a clearer program structure and higher modularity by the clean separation between algorithmic and behavioural code. There are two possible ways for applying the same techniques to strict languages:

- Use Haskell with evaluation strategies as an embedding coordination language. The top-level parallelism is specified in Haskell, the sequential components are written in a strict language. Although one of the main advantages of evaluation strategies over other coordination languages is the use of the same language for describing computation and coordination, a separation may be worthwhile for parallelising large programs written in a strict language.

- Implement a strategies module in the strict language based on non-strict data structures, which can be modelled in the strict language. These non-strict data structures can then be used in combining the parallel components of the code, leaving most of the code unchanged. This approach requires that all synchronisation is performed via non-strict data structures.

Although our visualisation tools provide important information about the parallel program behaviour we have noticed several shortcomings when using them on large programs. Most importantly it is not possible to link points in the activity profile to expressions or strategies in the source code. Therefore, it is sometimes hard to interpret an activity profile of a complex program. This observation has recently led to new research on parallel cost centre profiling (Hammond et al. 1997), to which the author is contributing. The idea here is to use cost centres as they have been developed for sequential profiling (Sansom & Peyton Jones 1995) and combine them with the GranSim simulator, yielding the GranCC parallel profiler. It is then possible to distinguish between threads that are currently evaluating expressions attached to different cost centres. Initial results with a first implementation have already provided further insights into the behaviour of some of our programs like Alpha-Beta search. Currently research is undertaken in order to augment the initial version of GranCC with a variant that links points in the activity profile with points in the behavioural rather than the algorithmic code. This would provide information about the parallelism generated by a certain strategy.

### Runtime-system

The granularity improvement mechanisms presented in Section 5.5 represent just a few possibilities how to exploit granularity information. More variants could be implemented, possibly providing different alternatives as options to the programmer. From the measurements presented in Section 5.6, mechanisms with a low overhead seem to be advantageous even if they do not make optimal use of the available information.

Other improvements and extensions could be made to the parallel runtime-system:

- Implementations of more runtime-system methods for improving granularity would be interesting. For example Aharoni et al. (1992) present a scheduling algorithm that guarantees that the parallel code performs no more than twice as many computations in total than the sequential code. This is done by enforcing a lower limit on the amount of computation that has to be performed by a thread before it is allowed to create other parallel threads. Using this idea in a production runtime-system rather than a prototype implementation would help to assess the practical usefulness of this algorithm.

- Based on the experiences with parallelising Lolita it would be useful to have a dynamically growable heap when running GUM, in particular on a shared memory machine. In the current version the heaps on all processors have to have the same size. This does not account for possible imbalance in heap usage. An implementation of dynamically growable heaps could use heap chunks similar to the currently used stack chunks, which are maintained as a list. The $\langle \nu, G \rangle$-machine, which has been designed for shared memory machines, uses a similar technique (Augustsson & Johnsson 1989).

Some experiments presented in this thesis also indicate that an important area for the efficient execution of parallel programs is the data locality in the program. In joint work the author has studied this issue in a comparison of various packing and rescheduling schemes in (Loidl & Hammond 1996*b*). However, this area clearly needs more work. In particular it might be advantageous to have annotations for explicit data placement or for transferring a data structure in its unevaluated form even if it already has been evaluated. In general it is not clear whether local or remote evaluation is better. A decision on a case by case basis, either via program annotations or inside the runtime-system, would be worth investigating.

Recent work on lazy threads (Goldstein et al. 1996) has achieved promising results in reducing the overhead attached to the bookkeeping of potential parallelism. In particular, measurements of a dataflow-based implementation on a CM-5 distributed memory machine showed significant speedups compared to a model that is closer related to the sparking model used in this thesis. Therefore, it would be interesting to study these techniques in the context of parallel graph reduction. The detailed measurements performed by Goldstein (1997) would be a good starting point for these evaluations.

## Analysis

The most immediate goal in extending the presented work should be an implementation of the static granularity analysis. For a more detailed evaluation of the quality of the analysis it would be necessary to apply it to a set of larger test programs. Starting from the detailed outline of an inference algorithm in Chapter 6, which is based on an existing implementation of sized types, a concrete implementation of the analysis for non-recursive expression should be straightforward. It would be close to the cost analysis for FX programs.

Several stages in the inference algorithm outlined in Section 6.5.1 need refinement in order to implement the full algorithm. The simplification algorithm has to merge symbolic cost and size functions in order to generate just one cost and size function for each user defined recursive function. Determining the costs for the base case of a recursive function requires in general a more sophisticated analysis of the head of conditionals in order to distinguish the recursion branch from the base case. Section 6.5.4 has given a first version of an algorithm for matching c-expressions with a library of recurrences, which is based on an unification approach. This algorithm most likely has to be refined in a concrete implementation. The specification of the library itself should mainly be a matter of tuning. From experiences of hand analysing programs and based on previous work, a small library should already capture a large class of recurrences. The final steps of the complete inference algorithm only perform syntactic checks on the structure of c-expression and define an interface to the Omega test, which is already provided by the existing implementation of type checking for sized types.

The most promising direction for extending the granularity analysis to lazy languages would be to develop a projections-based demand analysis, similar to the strictness analysis in Kubiak et al. (1991), and to use the derived information in order to extend the sized time system to lazy languages. This work could build on top of the cost calculus for lazy languages developed by Sands (1990*b*), which also uses projections. In this approach projection transformers have to be defined via a backward analysis in order to propagate demand through user defined functions. Only if the propagated projection requires the evaluation of an expression is the cost for the evaluation added to the total costs. Compared to the analysis of a strict language, the result is a weaker upper bound. Furthermore, it could be improved by having sharing information available. Therefore, an integration of strictness, sharing, and granularity analysis would be an interesting avenue of further work.

Another interesting piece of future work would be to study whether the library approach of Chapter 6 could be reused for other analyses. In general it should be possible to use it for any analysis over an integer domain. In fact in the sized time system the same machinery is used for performing size and cost analysis. The advantages of this approach, such as tunability via the size of the library and no restriction on the height of the domain, might make this an interesting alternative to abstract interpretation in general.

Replacing the library approach with a general recurrence solver probably yields a too expensive on-line analysis. However, for an off-line approach, where the granularity analysis is not part of the compilation process but only done rarely for optimising the parallel code, this approach might be feasible. Based on existing recurrence solvers in computer algebra systems it should be possible to implement an algorithm that covers most cases without being prohibitively expensive.

This thesis only outlines the structure of a soundness proof of the size and cost reconstruction algorithm. In order to assure that no wrong information is passed to the runtime-system a rigorous proof would be necessary. Furthermore, a dynamic semantics of $\mathcal{L}$ should be given in order to formalise the notion of computation steps and to show that the inference system describes theses costs.

As a simplified version of the granularity analysis discussed in Chapter 6 another analysis for inferring monotonicity information could be useful. The idea of such an analysis is to infer whether the cost function associated to a user defined function is monotonically increasing, decreasing, or neither. Based on the result and on knowledge about relative sizes of values it would be possible to infer relative costs between different calls to the same function. Although this yields only a partial order of costs the resulting information might be sufficient to yield some improvement in the performance of the program.

The ultimate goal of the work presented in this thesis is entirely implicit parallelism for GpH. In order to drive further research in this direction it is necessary to combine existing strictness, sharing and granularity analyses to obtain a system with genuine implicit parallelism. Probably this would reveal the necessity of further improvements in the runtime-system and of more accurate information provided by the analyses. As the experience from sequential compiler optimisations shows, an integration of all analyses and runtime-system methods into one system is essential to study interactions between the different improvements.

# Bibliography

Abramski, S. & Sykes, R. (1985), Secd-m: A Virtual Machine for Applicative Programming, *in* FPCA'85 — Conference on Functional Programming Languages and Computer Architecture, LNCS 201, Springer-Verlag, Nancy, France, September 16–19, pp. 81–98. (p 27)

Achten, P. (1991), Annotations for Load Distribution, *in* IFL'91 — International Workshop on the Parallel Implementation of Functional Languages, Technical Report CSTR 91-07, University of Southampton, UK, June 5–7. (p 183)

Aditya, S., Arvind & Maessen, J.-W. (1995), Semantics of pH: A Parallel Dialect of Haskell, CSG Memo 377-1, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA. (pp. 26, 30)
**URL:** `ftp://csg-ftp.lcs.mit.edu/pub/papers/csgmemo/memo-377-1.ps.gz`

Aharoni, G., Feitelson, D. & Barak, A. (1992), A Run-Time Algorithm for Managing the Granularity of Parallel Functional Programs, *Journal of Functional Programming* **2**(4), 387–405. (pp. 177, 238)

Aiken, A., Wimmers, E. & Lakshman, T. (1994), Soft Typing with Conditional Types, *in* POPL'94 — Symposium on Principles of Programming Languages, Orlando, FL, USA, January 10–13. (pp. 190, 199, 203)
**URL:** `http://http.cs.berkeley.edu/~aiken/ftp/popl94.ps`

Arvind, Caro, A., Maessen, J.-W. & Aditya, S. (1996), A Multithreaded Substrate and Compilation Model for the Implicitly Parallel Language pH, *in* LCPC-96. Also: CSG Memo 382. (p 32)
**URL:** `ftp://csg-ftp.lcs.mit.edu/pub/papers/csgmemo/memo-382.ps.gz`

Arvind, Nikhil, R. & Pingali, K. (1989), I-structures: Data Structures for Parallel Computing, *ACM Transactions on Programming Languages and Systems* **11**(4), 598–632. (pp. 32, 38)

Arvind & Nikhil, R. (1990), Executing a Program on the MIT Tagged-Token Dataflow Architecture, *IEEE Transactions on Computers* **39**(3). (pp. 8, 30, 181)

Augustsson, L. & Johnsson, T. (1989), Parallel Graph Reduction with the $\langle v, G\rangle$-machine, *in* FPCA'89 — Conference on Functional Programming Languages and Computer Architecture, ACM Press, Imperial College, London, UK, September 11–13, pp. 202–213. (pp. 35, 36, 41, 43, 239)
**URL:** `ftp://ftp.cs.chalmers.se/pub/cs-reports/papers/nu-G.ps.Z`

Augustsson, L. (1987), Compiling Lazy Functional Languages, Part II, PhD thesis, Department of Computer Sciences, Chalmers University of Technology, Göteborg, Sweden. (p 29)

Banâtre, J.-P. & Le Métayer, D. (1990), The Gamma Model and its Discipline of Programming, *Science of Programming* **15**(1), 55–77. (p 32)

Barendregt, H., van Eekelen, M., Hartel, P., Hertzberger, L., Plasmeijer, M. & Vree, W. (1987), The Dutch Parallel Reduction Machine Project, *Future Generation Computer Systems* **3**, 261–270. (pp. 40, 181)
**URL:** `ftp://ftp.cs.kun.nl/pub/CSI/SoftwEng.FunctLang/papers/-barh87-PRMprojekt.ps.gz`

Barth, P., Nikhil, R. & Arvind (1991), M-Structures: Extending a Parallel, Non-strict, Functional Language with State, *in* FPCA'91 — Conference on Functional Programming Languages and Computer Architectures, LNCS 523, Springer-Verlag, Harvard, MA, USA, pp. 538–568. (p 32)

Bennett, A. (1993), Parallel Graph Reduction for Shared-Memory Architectures, PhD thesis, Department of Computing, Imperial College, London. (p 54)

Bevan, D. (1987), Distributed Garbage Collection Using Reference Counting, *in* PARLE'87 — Parallel Architectures and Languages Europe, LNCS 259, Springer-Verlag, Eindhoven, The Netherlands, June 12–16, pp. 176–187. (p 51)

Bird, R. & Wadler, P. (1988), *Introduction to Functional Programming*, Prentice Hall. (p 22)

Bjerner, B. & Holmström, S. (1989), A Compositional Approach to Time Analysis of First Order Lazy Functional Programs, *in* FPCA'89 — Conference on Functional Programming Languages and Computer Architecture, ACM Press, Imperial College, London, UK, September 11–13, pp. 157–165. (p 227)

Bjerner, B. (1989), Time Complexity of Programs in Type Theory, PhD thesis, Department of Computer Sciences, University of Göteborg. (pp. 226, 227)

Blelloch, G. & Narlikar, G. (1994), A Practical Comparison of $N$-Body Algorithms, *in* Parallel Algorithms, American Mathematical Society. (pp. 153, 155)
**URL:** `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/scandal/public/-papers/dimacs-nbody.ps.gz`

Blelloch, G. (1996), Programming Parallel Algorithms, *Communications of the ACM* **39**(3), 85–97. (pp. 26, 33)
**URL:** `http://www.cs.cmu.edu/~scandal/cacm.html`

Bloss, A. (1989), Path Analysis and the Optimization of Non-strict Functional Languages, Research report YALEU/DCS/RR-704, Department of Computer Science, Yale University. (p 226)

Blumofe, R., Joerg, C., Kuszmaul, B., Leiserson, C., Randall, K. & Zhou, Y. (1995), Cilk: An Efficient Multithreaded Runtime System, *in* PPoPP'95 — Symposium on Principles and Practice of Parallel Programming, Santa Barbara, CA, USA, July 19–21, pp. 207–216. (p 177)
**URL:** `ftp://theory.lcs.mit.edu/pub/cilk/PPoPP95.ps.Z`

Bobrow, D. & Wegbreit, B. (1973), A Model and Stack Implementation of Multiple Environments, *Communications of the ACM* **16**, 591–602. (p 44)

Böhm, A., Cann, D., Feo, J. & Oldehoeft, R. (1991), SISAL 2.0 Reference Manual, Technical Report CS-91-118, Computer Science Department, Colorado State University. (pp. 26, 30)

Botorog, G. & Kuchen, H. (1996), Skil: An Imperative Language with Algorithmic Skeletons for Efficient Distributed Programming, *in* HPDC'96 — International Symposium on High Performance Distributed Computing, IEEE Computer Society Press, pp. 243–252. (p 146)
**URL:** `http://www-i2.informatik.rwth-aachen.de/botorog/Papers/-hpdc96.ps.gz`

Bratvold, T. (1994), Skeleton-Based Parallelisation of Functional Programs, PhD thesis, Department of Computing and Electrical Engineering, Heriot-Watt University, Edinburgh. (p 184)
**URL:** `ftp://ftp.cee.hw.ac.uk/pub/funcprog/tab.phd.ps.Z`

Brodal, G. & Okasaki, C. (1996), Optimal Purely Functional Priority Queues, *Journal of Functional Programming* **6**(6), 839–857. (p 170)
**URL:** `http://foxnet.cs.cmu.edu/people/cokasaki/priority.ps`

Brodal, G. (1996), Worst-Case Priority Queues, *in* SODA'96 — Symposium on Discrete Algorithms, ACM SIAM, pp. 52–58. (p 170)
**URL:** `http://www.mpi-sb.mpg.de/~brodal/Papers/soda96.ps.gz`

Bülck, T., Held, A., Kluge, W., Pantke, S., Rathsack, A., Scholz, S.-B. & Schröder, R. (1994), Experience with the Implementation of a Concurrent Graph Reduction System on an nCube/2 Platform, *in* CONPAR'94 — Conference on Algorithms and Hardware for Parallel Processing, LNCS 854, Springer-Verlag, Linz, Austria, September 6–8, pp. 497–508. (p 50)

Burge, W. H. (1975), *Recursive Programming Techniques*, Addison-Wesley, Reading, MA, USA. (p 27)

Burn, G., Peyton Jones, S. & Robson, J. (1988), The Spineless G-machine, *in* LFP'88 — Conference on Lisp and Functional Programming, Salt Lake City, UT, USA, pp. 244–258. (p 29)

Burn, G. (1987), Evaluation Transformers — A Model for the Parallel Evaluation of Functional Languages (Extended Abstract), *in* FPCA'87 — Conference on Functional Programming Languages and Computer Architecture, LNCS 274, Springer-Verlag, Portland, OR, USA, September 14–16, pp. 446–470. (pp. 21, 26)

Burn, G. (1990), Using Projection Analysis in Compiling Lazy Functional Programs, *in* LFP'90 — Conference on Lisp and Functional Programming, ACM Press, Nice, France, June 27–29, pp. 227–241. (p 226)

Burn, G. (1991*a*), Implementing the Evaluation Transformer Model of Reduction on Parallel Machines, *Journal of Functional Programming* **1**(3), 329–366. (pp. 21, 226)
**URL:** `http://theory.doc.ic.ac.uk/tfm/papers/BurnGL/-JnlFP_Implementation.ps.gz`

Burn, G. (1991*b*), *Lazy Functional Languages: Abstract Interpretation and Compilation*, Research Monographs in Parallel and Distributed Computing, Pitman. (p 21)

Burton, F. & Sleep, M. (1981), Executing Functional Programs on a Virtual Tree of Processors, *in* FPCA'81 — Conference on Functional Programming Languages and Computer Architecture, Portsmouth, NH, USA, pp. 187–194. (pp. 33, 181)

Burton, F. (1984), Annotations to Control Parallelism and Reduction Order in the Distributed Evaluation of Functional Programs, *ACM Transactions on Programming Languages and Systems* **6**(2), 159–174. (pp. 26, 147)

Busvine, D. (1993), Detecting Parallel Structures in Functional Programs, PhD thesis, Department of Computing and Electrical Engineering, Heriot-Watt University, Edinburgh. (p 184)
**URL:** `ftp://ftp.cee.hw.ac.uk/pub/funcprog/djb.phd.tar.Z`

Char, B., Geddes, K., Gonnet, G., Leong, B., Monagan, M. & Watt, S. (1991), *Maple V — Library Reference Manual*, Springer-Verlag. (p 213)

Chiou, D., Ang, B. A., Beckerle, M., Boughton, G., Greiner, R., Hicks, J. & Hoe, J. (1995), StarT-NG: Delivering Seamless Parallel Computing, *in* EuroPar'95

— European Conference on Parallel Processing, LNCS 966, Springer-Verlag, Stockholm, Sweden, August 29–31, pp. 101–116. (pp. 30, 38, 47)

Church, A. (1941), *The Calculi of Lambda Conversion*, Princeton University Press. (p 19)

Clack, C. & Peyton Jones, S. (1986), The Four-Stroke Reduction Engine, *in* LFP'86 — Conference on Lisp and Functional Programming, ACM Press, Cambridge, MA, USA, August 4–6, pp. 220–232. (p 35)

Cole, M. (1989), *Algorithmic Skeletons: Structured Management of Parallel Computation*, MIT Press. (pp. 23, 26, 145, 184)

Cousot, P. & Cousot, R. (1977), Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, *in* POPL'77 — Symposium on Principles of Programming Languages, Los Angeles, CA, USA, pp. 238–252. (p 190)

Culler, D., Goldstein, S., Schauser, K. & von Eicken, T. (1993), TAM — A Compiler Controlled Threaded Abstract Machine, *Journal of Parallel and Distributed Computing* **18**, 347–370. (pp. 30, 38, 39, 47)
**URL:** `http://www.cs.ucsb.edu/~schauser/papers/93-jpdc-tr.ps`

Culler, D. (1990), Managing Parallelism and Resources in Scientific Dataflow Programs, PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA. (p 182)

Curien, P.-L. (1986), *Categorical Combinators, Sequential Algorithms and Functional Programming*, Pitman. (p 27)

Curry, H. & Feys, R. (1958), *Combinatory Logic*, Vol. 1, North Holland. (p 27)

Darlington, J., Guo, Y., To, H. & Yang, J. (1995), Functional Skeletons for Parallel Coordination, *in* EuroPar'95 — European Conference on Parallel Processing, LNCS 966, Springer-Verlag, Stockholm, Sweden, August 29–31. (pp. 147, 154, 184)
**URL:** `http://www-ala.doc.ic.ac.uk/papers/Y.Guo/europar.ps.Z`

Darlington, J. & Reeve, M. (1981), ALICE — A Multi-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages, *in* FPCA '81 — Conference on Functional Programming Languages and Computer Architecture, ACM Press, Boston, MA, USA, pp. 65–74. (pp. 8, 39, 42)

Davie, A. & McNally, D. (1990), CASE - A Lazy Version of an SECD Machine with a Flat Environment, Technical Report CS/90/19, University of St Andrews. (p 27)
**URL:** `ftp://ftp.dcs.st-and.ac.uk/pub/staple/CASE.ps.Z`

Davis, K. (1994), Projection-based Program Analysis, PhD thesis, Department of Computing Science, University of Glasgow. (p 226)

Debbabi, M., Aïdoud, Z. & Faour, A. (1997), On the Inference of Structured Recursive Effects with Subtyping, *Journal of Functional and Logic Programming* **1997**(5). (p 207)
**URL:** `http://www.cs.tu-berlin.de/journal/jflp/articles/1997/A97-05/-JFLP-A97-05.ps.gz`

Debray, S., Lin, N.-W. & Hermenegildo, M. (1990), Task Granularity Analysis in Logic Programs, *in* PLDI'90 — Programming Languages Design and Implementation, Vol. 25(6) of *SIGPLAN Notices*, ACM Press, White Plains, NY, USA, June 20–22, pp. 174–188. (p 227)

Debray, S. & Lin, N.-W. (1993), Cost Analysis of Logic Programs, *ACM Transactions on Programming Languages and Systems* **15**(5), 826–875. (p 227)

Debray, S., López García, P., Hermenegildo, M. & Lin, N.-W. (1994), Estimating the Computational Costs of Logic Programs, *in* SAS'94 — Static Analysis Symposium, LNCS 864, Springer-Verlag, Namur, Belgium, pp. 255–265. (p 227)

Dennis, J. (1974), First Version of a Data Flow Procedure Language, *in* Programming Symposium, LNCS 19, Springer-Verlag, Paris. (p 29)

Deschner, J. (1989), Simulating Multiprocessor Architectures for Compiled Graph-Reduction, *in* Glasgow Workshop on Functional Programming, Workshops in Computing, Springer-Verlag, Fraserburgh, Scotland, UK, August 21–23, pp. 225–237. (p 54)

Dornic, V. (1993), Ordering Times, Research Report YALEU/DCS/RR-956, Department of Computer Science, Yale University. (p 224)

Draghicescu, M. & Purushothaman, S. (1990), A Compositional Analysis of Evaluation Order and its Applications, *in* LFP'90 — Conference on Lisp and Functional Programming, ACM Press, Nice, France, June 27–29, pp. 242–250. (p 226)

Field, A. & Harrison, P. (1988), *Functional Programming*, Addison-Wesley. (pp. 19, 20, 27, 205)

Flajolet, P., Salvy, B. & Zimmermann, P. (1991), Automatic Average-Case Analysis of Algorithms, *Theoretical Computer Science* **79**, 37–109. (p 223)

Flanagan, C. & Nikhil, R. (1996), pHluid: The Design of a Parallel Functional Language Implementation on Workstations, *in* ICFP'96 — International Conference on Functional Programming, ACM Press, Philadelphia, PA, USA, May 24–26, pp. 169–179. (pp. 38, 47)

Foster, I. & Taylor, S. (1994), A Compiler Approach to Scalable Concurrent-Program Design, *ACM Transactions on Programming Languages and Systems* **16**(3), 577–604. (p 146)

Freeh, V. & Andrews, G. (1995), `fsc`: A Sisal Compiler for Both Distributed- and Shared-Memory Machines, *in* HPFC'95 — High Performance Functional Computing, Denver, CO, USA, April 10–12. (p 34)
**URL:** `ftp://sisal.llnl.gov/pub/hpfc/papers95/paper20.ps`

Gelernter, D. & Carriero, N. (1992), Coordination Languages and Their Significance, *Communications of the ACM* **32**(2), 97–107. (pp. 146, 147)

Gladitz, K. & Kuchen, H. (1996), Shared Memory Implementation of the Gamma-Operation, *Journal of Symbolic Computation* **21**, 577–591. (p 33)
**URL:** `http://www-i2.informatik.rwth-aachen.de/~herbert/jsc95.ps`

Goldberg, B. (1988*a*), Multiprocessor Execution of Functional Programs, PhD thesis, Department of Computer Science, Yale University. (pp. 19, 42, 163, 189, 222)

Goldberg, B. (1988*b*), Multiprocessor Execution of Functional Programs, *International Journal of Parallel Programming* **17**(5), 425–473. (pp. 38, 42, 49)

Goldsmith, R., McBurney, D. & Sleep, M. (1993), *Term Graph Rewriting: Theory and Practice*, John Wiley & Sons, chapter Parallel Execution of Concurrent Clean on ZAPP. (p 33)

Goldstein, S., Schauser, K. & Culler, D. (1996), Lazy Threads: Implementing a Fast Parallel Call, *Journal of Parallel and Distributed Computing* **37**(1), 5–20. (pp. 35, 57, 180, 239)
**URL:** `http://http.cs.berkeley.edu/~sethg/papers/jpdc.ps.Z`

Goldstein, S. (1997), Lazy Threads: Compiler and Runtime Structures for Fine-Grained Parallel Programming, PhD thesis, University of California, Berkeley. (pp. 180, 239)

Gomard, C. K. & Sestoft, P. (1991), Evaluation Order Analysis for Lazy Data Structures, *in* Glasgow Workshop on Functional Programming, Workshops in Computing, Springer-Verlag, Isle of Skye, Scotland, UK, August 13–15, pp. 112–127. (p 226)

Grant, P., Sharp, J., Webster, M. & Zhang, X. (1995), Experiences of Parallelizing Finite-Element Problems in a Functional Style, *Software – Practice and Experience* **25**(9), 947–974. (p 152)

GranSim (1998), GranSim Home Page, WWW page. (p 56)
**URL:** `http://www.dcs.glasgow.ac.uk/fp/software/gransim/default.html`

Greiner, J. (1994), A Comparison of Data-Parallel Algorithms for Connected Components, *in* SPAA'94 — Symposium on Parallel Algorithms and Architectures, Cape May, NJ, USA., pp. 16–25. Also: Technical Report CMU-CS-93-191. (p 155)
**URL:** `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/scandal/public/-papers/concomp-spaa94.ps.gz`

Gremban, K., Miller, G. & Zagha, M. (1994), Performance Evaluation of a New Parallel Preconditioner, Technical Report CMU-CS-94-205, School of Computer Science, Carnegie Mellon University. (p 155)
**URL:** `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/scandal/public/-papers/CMU-CS-94-205.ps.gz`

Gurd, J., Kirkham, C. & Watson, I. (1985), The Manchester Prototype Dataflow Computer, *Communications of the ACM* **28**(1), 34–52. (pp. 30, 49)

Haines, M. & Böhm, W. (1992), Software Multithreading in a Conventional Distributed Memory Multiprocessor, Technical Report CS-92-126, Colorado State University. (p 34)

Hall, C., Loidl, H.-W., Trinder, P., Hammond, K. & O'Donnell, J. (1997), Refining a Parallel Algorithm for Calculating Bowings, *in* Glasgow Workshop on Functional Programming, Ullapool, Scotland, UK, September 15–17. Submitted for publication. (pp. 99, 141, 143, 176)

Halstead, Jr., R. (1985), Multilisp: A Language for Concurrent Symbolic Computation, *ACM Transactions on Programming Languages and Systems* **7**(4), 501–538. (p 26)

Halstead Jr., R. (1995), Understanding the Performance of Parallel Symbolic Programs, *in* PSLS'95 — International Workshop on Parallel Symbolic Languages and Systems, LNCS 1068, Springer-Verlag, Beaune, France, pp. 81–107. (p 68)

Hammes, J., Lubeck, O. & Böhm, W. (1995), Comparing Id and Haskell in a Monte Carlo photon transport code, *Journal of Functional Programming* **5**(3), 283–316. (p 154)
**URL:** `http://www.cs.colostate.edu/~hammes/documents/final1.ps.Z`

Hammond, K., Loidl, H.-W. & Partridge, A. (1995), Visualising Granularity in Parallel Programs: A Graphical Winnowing System for Haskell, *in* HPFC'95 — High Performance Functional Computing, Denver, CO, USA, April 10–12, pp. 208–221. (pp. 11, 66, 74, 75, 143, 235)
**URL:** `http://www.dcs.st-and.ac.uk/~kh/papers/hpfc95/hpfc95.html`

Hammond, K., Loidl, H.-W. & Trinder, P. (1997), Parallel Cost Centre Profiling, *in* Glasgow Workshop on Functional Programming, Ullapool, Scotland, UK, September 15–17. Submitted for publication. (pp. 68, 82, 238)

Hammond, K., Mattson Jr., J. & Peyton Jones, S. (1994), Automatic Spark Strategies and Granularity for a Parallel Functional Language Reducer, *in* CONPAR'94 — Conference on Algorithms and Hardware for Parallel Processing, LNCS 854, Springer-Verlag, Linz, Austria, September 6–8, pp. 521–532. (pp. 8, 50, 85, 162)
**URL:** `ftp://ftp.dcs.glasgow.ac.uk/pub/glasgow-fp/papers/-spark-strategies-and-granularity.ps.Z`

Hammond, K. & Peyton Jones, S. (1990), Some Early Experiments on the GRIP Parallel Reducer, *in* IFL'90 — International Workshop on the Parallel Implementation of Functional Languages, Nijmegen, The Netherlands, pp. 51–72. (p 60)

Hammond, K. & Peyton Jones, S. (1992), Profiling Scheduling Strategies on the GRIP Multiprocessor, *in* IFL'92 — International Workshop on the Parallel Implementation of Functional Languages, RWTH Aachen, Germany, September 28–30, pp. 73–98. (pp. 8, 50)
**URL:** `ftp://ftp.dcs.gla.ac.uk/pub/glasgow-fp/papers/-grip-scheduling.ps.gz`

Hammond, K. (1993), Getting a GRIP, *in* IFL'93 — International Workshop on the Parallel Implementation of Functional Languages, Nijmegen, The Netherlands. (p 55)
**URL:** `ftp://ftp.dcs.gla.ac.uk/pub/glasgow-fp/authors/Kevin_Hammond/-Getting_a_GRIP.ps.Z`

Hammond, K. (1994), Parallel Functional Programming: An Introduction, *in* PASCO'94 — International Symposium on Parallel Symbolic Computation, World Scientific, Hagenberg/Linz, Austria, September 26–28, pp. 181–193. (p 26)
**URL:** `http://www.dcs.st-and.ac.uk/~kh/papers/pasco94/pasco94.html`

Harrison, P. & Reeve, M. (1986), The Parallel Graph Reduction Machine, ALICE, *in* Workshop on Graph Reduction, LNCS 279, Springer-Verlag, Santa Fe, NM, USA, pp. 181–202. (pp. 42, 48, 49)

Hartel, P., Hofman, R., Langendoen, K., Muller, H., Vree, W. & Hertzberger, L. (1995), A Toolkit for Parallel Functional Programming, *Concurrency — Practice and Experience* **7**(8), 765–793. (pp. 144, 152, 153)
**URL:** `ftp://ftp.fwi.uva.nl/pub/computer-systems/functional/reports/-CPE_toolkit.ps.Z`

Hartel, P. (1995), Benchmarking Implementations of Functional Languages II — Two Years Later, *in* IFL'95 — International Workshop on the Implementation of Functional Languages, Båstad, Sweden, pp. 63–68. (p 77)
**URL:** ftp://ftp.fwi.uva.nl/pub/computer-systems/functional/reports/-benchmarkII.ps.Z

Henglein, F. (1993), Type Inference with Polymorphic Recursion, *ACM Transactions on Programming Languages and Systems* **15**(2), pp. 253–289. (p 196)

Hermenegildo, M. & López García, P. (1995), Efficient Term Size Computation for Granularity Control, *in* International Conference on Logic Programming, MIT Press, pp. 647–661. (p 228)

Hickey, T. & Cohen, J. (1988), Automating Program Analysis, *Journal of the ACM* **35**(1), 185–220. (p 223)

Hofman, R., Langendoen, K. & Vree, W. (1992), Scheduling Consequences of Keeping Parents at Home, *in* ICPADS'92 — Parallel and Distributed Systems, National Tsing Hwa University, Taiwan, pp. 580–588. (p 40)

Hofman, R. (1994), Scheduling and Grain Size Control, PhD thesis, Department of Computer Systems, University of Amsterdam. (pp. 54, 181)
**URL:** ftp://ftp.fwi.uva.nl/pub/computer-systems/functional/reports/-thesis_Hofman.ps.Z

Hogen, G. & Loogen, R. (1994), Efficient Organization of Control Structures in Distributed Implementations, *in* CC'94 — International Conference on Compiler Construction, LNCS 786, Springer-Verlag, pp. 98–112. (pp. 36, 44, 46)
**URL:** http://www-i2.informatik.rwth-aachen.de/OldStaff/hogen/-PUBLICATIONS/cc94.ps.gz

Hogen, G. & Loogen, R. (1995), Parallel Functional Implementations: Graphbased vs. Stackbased Reduction, *in* Fuji International Workshop on Functional and Logic Programming, World Scientific. (p 44)
**URL:** ftp://ftp.informatik.rwth-aachen.de/pub/reports/1994/94-18.ps.gz

Hong, H. & Loidl, H.-W. (1994), Parallel Computation of Modular Multivariate Polynomial Resultants on a Shared Memory Machine, *in* CONPAR'94 — Conference on Parallel and Vector Processing, LNCS 854, Linz, Austria, September 6–8, pp. 325–336. (p 141)
**URL:** http://www.dcs.gla.ac.uk/~hwloidl/publications/resultant.ps.gz

Hong, H., Schreiner, W., Neubacher, A., Siegl, K., Loidl, H.-W., Jebelean, T. & Zettler, P. (1992), PACLIB User Manual, Technical Report 92-32, RISC-Linz, Johannes Kepler University, Linz, Austria. (p 139)

Hudak, P. & Goldberg, B. (1985), Serial Combinators: Optimal Grains of Parallelism, *in* FPCA'85 — Conference on Functional Programming Languages and Computer Architecture, LNCS 201, Springer-Verlag, Nancy, France, September 16–19, pp. 382–388. (p 182)

Hudak, P. (1986), Para-Functional Programming, *IEEE Computer* **19**(8), 60–70. (pp. 148, 182)

Hudak, P. (1991), Para-Functional Programming in Haskell, *in* B. Szymanski, ed., Parallel Functional Languages and Compilers, ACM Press and Addison-Wesley. (pp. 149, 182)

Huelsbergen, L., Larus, J. & Aiken, A. (1994), Using Run-Time List Sizes to Guide Parallel Thread Creation, *in* LFP'94 — Conference on Lisp and Functional Programming, ACM Press, Orlando, FL, USA, June 27–29, pp. 79–90. (p 224)

Hughes, R., Pareto, L. & Sabry, A. (1996), Proving the Correctness of Reactive Systems Using Sized Types, *in* POPL'96 — Symposium on Principles of Programming Languages, ACM Press, St Petersburg, FL, USA, January 21–24. (pp. 12, 188, 191, 196, 200, 204, 225, 228, 231, 236)
**URL:** `http://www.cs.chalmers.se/~rjmh/Papers/popl-96.ps`

Hughes, R. (1984), The Design and Implementation of Programming Languages, PhD thesis, Programming Research Group, University of Oxford. (p 27)

Hughes, R. (1987), Analysing Strictness by Abstract Interpretation of Continuations, *in* S. Abramsky & C. Hankin, eds, Abstract Interpretation of Declarative Languages, Ellis Horwood, pp. 63–102. (p 226)

Hughes, R. (1989), Why Functional Programming Matters, *The Computer Journal* **32**(2), 98–107. (pp. 91, 104, 105, 110)
**URL:** `http://www.cs.chalmers.se/~rjmh/Papers/whyfp.ps`

Hwang, S. & Rushall, D. (1992), The $\nu$-STG machine: A Parallelized Spineless Tagless Graph Reduction Machine in a Distributed Memory Architecture (Draft Version), *in* IFL'92 — International Workshop on the Parallel Implementation of Functional Languages, RWTH Aachen, Germany, September 28–30. (pp. 35, 46)

Ito, N., Sato, M., Kishi, A., Kuno, E. & Rokusawa, K. (1986), The Architecture and Preliminary Evaluation Results of the Experimental Parallel Inference Machine PIM-D , *in* ISCA'86 — International Symposium on Computer Architecture, IEEE Computer Society, Tokyo, Japan, June 2–5, pp. 149-156. (p 30)

Jay, C., Cole, M., Sekanina, M. & Steckler, P. (1997), A Monadic Calculus for Parallel Costing of a Functional Language of Arrays, *in* EuroPar'97 — European Conference on Parallel Processing, LNCS 1300, Springer-Verlag, Passau, Germany, pp. 650–661. (p 223)

Johnsson, T. (1985), Lambda Lifting: Transforming Programs to Recursive Equations, *in* FPCA'85 — Conference on Functional Programming Languages and Computer Architecture, LNCS 201, Springer-Verlag, Nancy, France, September 16–19, pp. 190–203. (p 27)

Johnsson, T. (1987), Compiling Lazy Functional Languages, Part I, PhD thesis, Department of Computer Sciences, Chalmers University of Technology, Göteborg, Sweden. (p 29)

Jones, S. & Le Métayer, D. (1989), Compile-Time Garbage Collection by Sharing Analysis, *in* FPCA'89 — Conference on Functional Programming Languages and Computer Architecture, ACM Press, Imperial College, London, UK, September 11–13, pp. 54–74. (p 36)

Jouvelot, P. & Gifford, D. (1991), Algebraic Reconstruction of Types and Effects, *in* POPL'91 — Symposium on Principles of Programming Languages, ACM Press, pp. 303–310. (p 204)
**URL:** `http://www.psrg.lcs.mit.edu/ftpdir/papers/popl91.dvi`

Joy, M. & Axford, T. (1992), A Parallel Graph Reduction Machine, *in* IFL'92 — International Workshop on the Parallel Implementation of Functional Languages, RWTH Aachen, Germany, September 28–30. (p 54)

Junaidu, S. (1998), A Parallel Functional Language Compiler for Message Passing Multicomputers, PhD thesis, School of Mathematical and Computational Sciences, University of St. Andrews. (pp. 79, 143)

Kaser, O., Pawagi, S., Ramakrishnan, C., Ramakrishnan, I. & Sekar, R. (1992), Fast Parallel Implementation of Lazy Languages — The EQUALS Experience, *in* LFP'92 — Conference on Lisp and Functional Programming, ACM Press, San Francisco, CA, USA, June 22–24, pp. 335–344. (p 177)

Kaser, O., Pawagi, S., Ramakrishnan, C., Ramakrishnan, I. & Sekar, R. (1997), EQUALS — a Fast Parallel Implementation of a Lazy Language, *Journal of Functional Programming* **7**(2), 183–217. (pp. 21, 36)

Keane, J. (1994), An Overview of the Flagship System, *Journal of Functional Programming* **4**(1), 19–45. (pp. 39, 46, 177)

Keller, R. & Lin, F. (1984), Simulated Performance of a Reduction-Based Multiprocessor, *IEEE Computer* **17**(7), 70–82. (p 54)

Kelly, P. (1989), *Functional Programming for Loosely-Coupled Multiprocessors*, Research Monographs in Parallel and Distributed Computing, MIT Press. (pp. 26, 85, 148)

Kesseler, M. (1996), The Implementation of Functional Languages on Parallel Machines with Distributed Memory, PhD thesis, University of Nijmegen. (pp. 37, 46, 47)

Kewley, J. & Glynn, K. (1989), Evaluation Annotations for Hope+, *in* Glasgow Workshop on Functional Programming, Workshops in Computing, Springer-Verlag, Fraserburgh, Scotland, UK, August 21–23, pp. 329–337. (pp. 26, 147, 183)

Kingdon, H., Lester, D. & Burn, G. (1991), The HDG-machine: a Highly Distributed Graph-Reducer for a Transputer Network, *The Computer Journal* **34**(4), 290–301. (pp. 21, 35, 39, 41, 43, 49)
**URL:** `http://theory.doc.ic.ac.uk/tfm/papers/BurnGL/HDGmachine.ps.gz`

Knuth, D. (1981), *The Art of Computer Programming*, Vol. II: Seminumerical Algorithms, 2nd edition, Addison-Wesley. (p 128)

Kranz, D., Halstead Jr., R. & Mohr, E. (1989), Mul-T: A High-Performance Parallel Lisp, *in* PLDI'91 — Programming Languages Design and Implementation, Vol. 24(7) of *SIGPLAN Notices*, Portland, OR, USA, June 21–23, pp. 81–90. (pp. 26, 178)

Kubiak, R., Hughes, R. & Launchbury, J. (1991), Implementing Projection Based Strictness Analysis, *in* Glasgow Workshop on Functional Programming, Workshops in Computing, Springer-Verlag, Isle of Skye, Scotland, UK, August 13–15, pp. 207–224. (pp. 226, 240)
**URL:** `http://www.cse.ogi.edu/~jl/Papers/implementing.ps`

Kuo, T.-M. & Mishra, P. (1989), Strictness Analysis: a New Perspective Based on Type Inference, *in* FPCA'89 — Conference on Functional Programming Languages and Computer Architecture, ACM Press, Imperial College, London, UK, September 11–13, pp. 260–272. (p 190)

Landin, P. J. (1964), The Mechanical Evaluation of Expressions, *The Computer Journal* **6**, 308–320. (pp. 16, 27)

Lauer, M. (1982), Computing by Homomorphic Images, *in* B. Buchberger, G. Collins, R. Loos & R. Albrecht, eds, Computer Algebra — Symbolic and Algebraic Computation, Springer-Verlag, pp. 139–168. (p 124)

Launchbury, J. & Baraki, G. (1996), Representing Demand by Partial Projections, *Journal of Functional Programming* **6**(4), 563–585. (p 226)

Lester, D. (1989), Stacklessness: Compiling Recursion for a Distributed Architecture, *in* FPCA'89 — Conference on Functional Programming Languages and Computer Architecture, ACM Press, Imperial College, London, UK, September 11–13, pp. 116–128. (p 43)

Le Métayer, D. (1988), ACE: An Automatic Complexity Evaluator, *ACM Transactions on Programming Languages and Systems* **10**(2), 248–266. (p 225)

Limongelli, C. & Loidl, H.-W. (1993), Rational Number Arithmetic by Parallel P-adic Algorithms, *in* ACPC'93 — Parallel Computation — International ACPC Conference, LNCS 734, Springer-Verlag, Gmunden, Austria, October 4–6, pp. 72–86. (p 143)
**URL:** `http://www.dcs.gla.ac.uk/~hwloidl/publications/p-adic.ps.gz`

Lipson, J. D. (1971), Chinese Remainder and Interpolation Algorithms, *in* SYMSAM'71 — Symposium on Symbolic and Algebraic Manipulation, Academic Press, pp. 372–391. (pp. 124, 128)

Loidl, H.-W. & Hammond, K. (1994), GRAPH for PVM: Graph Reduction for Distributed Hardware, *in* IFL'94 — International Workshop on the Implementation of Functional Languages, University of East Anglia, Norwich, UK, September 7–9. (pp. 55, 236)
**URL:** `http://www.dcs.gla.ac.uk/~hwloidl/publications/IFL94.ps.gz`

Loidl, H.-W. & Hammond, K. (1995), On the Granularity of Divide-and-Conquer Parallelism, *in* Glasgow Workshop on Functional Programming, Workshops in Computing, Springer-Verlag, Ullapool, Scotland, UK, July 8–10. (pp. 12, 174, 237)
**URL:** `http://www.dcs.gla.ac.uk/~hwloidl/publications/GlaFp95.ps.gz`

Loidl, H.-W. & Hammond, K. (1996*a*), A Sized Time System for a Parallel Functional Language, *in* Glasgow Workshop on Functional Programming, Ullapool, Scotland, UK, July 8–10. (pp. xviii, 12, 236)
**URL:** `http://www.dcs.glasgow.ac.uk/fp/workshops/fpw96/Loidl.ps.gz`

Loidl, H.-W. & Hammond, K. (1996*b*), Making a Packet: Cost-Effective Communication for a Parallel Graph Reducer, *in* IFL'96 — International Workshop on the Implementation of Functional Languages, LNCS 1268, Springer-Verlag, Bad Godesberg, Germany, pp. 184–199. (pp. 9, 13, 61, 63, 64, 65, 236, 239)
**URL:** `http://www.dcs.gla.ac.uk/~hwloidl/publications/IFL96.ps.gz`

Loidl, H.-W., Morgan, R., Trinder, P., Poria, S., Cooper, C., Peyton Jones, S. & Garigliano, R. (1997), Parallelising a Large Functional Program; Or: Keeping LOLITA Busy, *in* IFL'97 — International Workshop on the Implementation of

Functional Languages, University of St. Andrews, Scotland, UK, September 10–12. To appear in LNCS. (pp. xviii, 91, 114)
**URL:** `http://www.dcs.gla.ac.uk/~hwloidl/publications/Lolita.ps.gz`

Loidl, H.-W. & Trinder, P. (1997), Engineering Large Parallel Functional Programs, *in* IFL'97 — International Workshop on the Implementation of Functional Languages, University of St. Andrews, Scotland, UK, September 10–12. To appear in LNCS. (pp. xviii, 11, 91, 105, 235)
**URL:** `http://www.dcs.gla.ac.uk/~hwloidl/publications/IFL97.ps.gz`

Loidl, H.-W. (1992), A Parallelizing Compiler for the Functional Programming Language EVE, Master's thesis, RISC-Linz, Johannes Kepler University, Linz, Austria. (p 158)

Loidl, H.-W. (1993), Solving a System of Linear Equations by Using a Modular Method, Technical Report 93-69, RISC-Linz, Johannes Kepler University, Linz, Austria. (p 139)

Loidl, H.-W. (1996), *GranSim User's Guide*, 0.03 edition, Department of Computing Science, University of Glasgow. (pp. 59, 68)
**URL:** `http://www.dcs.glasgow.ac.uk/fp/software/gransim/user_toc.html`

Loidl, H.-W. (1997), LinSolv: a Case Study in Strategic Parallelism, *in* Glasgow Workshop on Functional Programming, Ullapool, Scotland, UK, September 15–17. Submitted for publication. (pp. xviii, 91)
**URL:** `http://www.dcs.gla.ac.uk/~hwloidl/publications/LinSolv.ps.gz`

Loogen, R., Kuchen, H. & Indermark, K. (1989), Distributed Implementation of Programmed Graph Reduction, *in* PARLE'89 — Conference on Parallel Architectures and Languages Europe, LNCS 365, Springer-Verlag, Eindhoven, The Netherlands, pp. 136-157. (pp. 21, 39, 43, 46, 49)

López García, P., Hermenegildo, M. & Debray, S. (1994), Towards Granularity Based Control of Parallelism in Logic Programs, *in* World Scientific, Hagenberg/Linz, Austria, 26–28 September, pp. 133–144. (p 227)
**URL:** `ftp://clip.dia.fi.upm.es/pub/papers/granul_control94.ps.Z`

López García, P., Hermenegildo, M. & Debray, S. (1995), A Methodology for Granularity Based Control of Parallelism in Logic Programs, *Journal of Symbolic Computation* **22**, 715–734. (p 227)

Lowenthal, D., Freeh, V. & Andrews, G. (1996), Using Fine-Grain Threads and Run-Time Decision-Making in Parallel Computing, *Journal of Parallel and Distributed Computing* **37**(1), 41–54. (p 177)
**URL:** `ftp://ftp.cs.arizona.edu/reports/1996/TR96-01.ps`

Lucassen, J. & Gifford, D. (1988), Polymorphic Effect Systems, *in* POPL'88 — Symposium on Principles of Programming Languages, ACM, San Diego, CA, USA, pp. 47–57. (p 207)

Maheshwari, P. (1995), Partitioning and Scheduling of Parallel Functional Programs for Larger Grain Execution, *Journal of Parallel and Distributed Computing* **26**(2), 151–165. (p 163)

Maranget, L. (1991), GAML: a Parallel Implementation of Lazy ML, *in* FPCA'91 — Conference on Functional Programming Languages and Computer Architectures, LNCS 523, Springer-Verlag, Harvard, MA, USA, pp. 102–123. (pp. 36, 177)

Marlow, S. & Wadler, P. (1997), A Practical Subtyping System for Erlang, *in* ICFP'97 — International Conference on Functional Programming, ACM Press, June 9–11, Amsterdam, The Netherlands, pp. 136–149. (p 199)
**URL:** `ftp://ftp.dcs.gla.ac.uk/pub/glasgow-fp/authors/Simon_Marlow/-erltc.ps`

McBurney, A. & Sleep, M. (1987), Transputer Based Experiments with the ZAPP Architecture, *in* PARLE'87 — Parallel Architectures and Languages Europe, LNCS 258, Springer-Verlag, Eindhoven, The Netherlands, June 12–16, pp. 242–259. (pp. 33, 49)

McColl, W. (1996), Scalability, Portability and Predictability: The BSP Approach to Parallel Programming, *Future Generation Computer Systems* **12**(4), 265–272. (p 17)

Michaelson, G., Ireland, A. & King, P. (1997), Towards a Skeleton Based Parallelising Compiler for SML, *in* IFL'97 — International Workshop on the Implementation of Functional Languages, University of St. Andrews, Scotland, UK, September 10–12. (p 184)
**URL:** `ftp://ftp.cee.hw.ac.uk/pub/funcprog/mik.ifl97.ps.Z`

Michaelson, G. & Scaife, N. (1995), Prototyping a Parallel Vision System in Standard ML, *Journal of Functional Programming* **5**(3), 345–382. (p 154)
**URL:** `ftp://ftp.cee.hw.ac.uk/pub/funcprog/ms.jfp95.ps.Z`

Milner, R. (1978), A Theory of Type Polymorphism in Programming Languages, *Journal of Computer and System Sciences* **17**, 348–375. (p 196)

Mirani, R. & Hudak, P. (1995), First-Class Schedules and Virtual Maps, *in* FPCA'95 — Conference on Functional Programming Languages and Computer Architecture, ACM Press, La Jolla, CA, USA, June 26–28, pp. 78–85. (pp. 26, 148, 151, 182)

Mitchell, J. (1991), Type Inference with Simple Subtypes, *Journal of Functional Programming* **1**(3), 245–285. (p 204)

Mohr, E., Kranz, D. & Halstead Jr., R. (1990), Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs, *in* LFP'90 — Conference on Lisp and Functional Programming, Nice, France, June 27–29, pp. 185–197. (pp. 166, 178)
**URL:** `ftp://crl.dec.com/pub/DEC/CRL/tech-reports/90.7.ps.Z`

Morais, D. (1986), Id World: An Environment for the Development of Dataflow Programs Written in Id, Technical Report MIT-LCS-TR-365, Laboratory of Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA. (p 54)

Morgan, R., Smith, M. & Short, S. (1994), Translation by Meaning and Style in Lolita, *in* International BCS Conference — Machine Translation Ten Years On, Cranfield University. (p 114)

Mycroft, A. (1984), Polymorphic Type Schemes and Recursive Definitions, *in* International Conference on Programming, LNCS 167, Springer-Verlag. (p 196)

Nikhil, R. (1989), The Parallel Programming Language Id and its Compilation for Parallel Machines, *in* Workshop on Massive Parallelism: Hardware, Programming and Applications, Amalfi, Italy. CSG Memo 313. (pp. 26, 30)

Nikhil, R. (1994), Cid: A Parallel "Shared-memory" C for Distributed Memory Machines, *in* Workshop on Languages and Compilers for Parallel Computing, LNCS 892, Springer-Verlag, Ithaca, NY, USA, August 1994, pp. 376–390. (p 177)
**URL:** `http://www.research.digital.com/CRL/personal/nikhil/cid/cid.ps.Z`

Nikhil, R. (1995), Parallel Symbolic Computing in Cid, *in* PSLS'95 — Workshop on Parallel Symbolic Computing, LNCS 1068, Springer-Verlag, Beaune, France, October 2–4, pp. 217–242. (p 177)
**URL:** `http://www.research.digital.com/CRL/personal/nikhil/cid/-cid_symbolic.ps.Z`

Nöcker, E., Plasmeijer, R. & Smetsers, S. (1991), The Parallel ABC Machine, *in* IFL'91 — International Workshop on the Parallel Implementation of Functional Languages, Technical Report CSTR 91-07, University of Southampton, UK, June 5–7, pp. 351–381. (pp. 35, 36, 44, 46)

Nöcker, E., Smetsers, J., van Eekelen, M. & Plasmeijer, M. (1991), Concurrent Clean, *in* PARLE'91 — Parallel Architectures and Languages Europe, LNCS 505, Springer-Verlag, Eindhoven, The Netherlands, pp. 202–219. (pp. 26, 145, 147, 183)

**URL:** ftp://ftp.cs.kun.nl/pub/CSI/SoftwEng.FunctLang/papers/-noce91-concurrentclean.ps.gz

NoFib (1998), The NoFib Benchmark Suite, WWW page. (p 67)
**URL:** http://www.dcs.gla.ac.uk/fp/software/ghc/nofib.html

Ostheimer, G. (1991), Parallel Functional Computation on STAR:DUST, *in* IFL'91 — International Workshop on the Parallel Implementation of Functional Languages, Technical Report CSTR 91-07, University of Southampton, UK, June 5–7, pp. 393–407. (p 50)
**URL:** ftp://ftp.dcs.st-and.ac.uk/pub/staple/stardust.ps.Z

Papadopoulos, G. & Culler, D. (1990), Monsoon: An Explicit Token-Store Architecture, *in* ISCA'90 — International Symposium on Computer Architecture, Vol. 18(2) of *ACM SIGARCH Computer Architecture News*, Seattle, WA, USA, May 28–31, pp. 82–91. (pp. 30, 38)

Park, Y. & Goldberg, B. (1992), Order-of-Demand Analysis for Lazy Languages, *in* WSA'92 — Analyse Statique, Bordeaux, France, September 23–25, pp. 91–101. (p 226)

Partain, W. (1992), The NoFib Benchmark Suite of Haskell Programs, *in* Glasgow Workshop on Functional Programming, Workshops in Computing, Springer-Verlag, Ayr, Scotland, UK, pp. 195–202. (p 67)

Pelagatti, S. (1993), A Methodology for the Development and the Support of Massively Parallel Programs, PhD thesis, Universita Delgi Studi Di Pisa. (p 147)

Peterson, J., Hammond, K. et al. (1996), *Haskell 1.3 — A Non-Strict, Purely Functional Language.* (pp. 24, 97)
**URL:** http://haskell.org/report/index.html

Petkovšek, M. & Salvy, B. (1993), Finding All Hypergeometric Solutions of Linear Differential Equations, *in* ISSAC'93 — International Symposium on Symbolic and Algebraic Computation, ACM Press, Kiev, Ukraine, July 6–8, pp. 27–33. (p 214)

Petkovšek, M. (1990), Finding Closed-Form Solutions of Difference Equations by Symbolic Methods, PhD thesis, School of Computer Science, Carnegie Mellon University. CMU-CS-91-103. (p 214)

Peyton Jones, S., Clack, C., Salkild, J. & Hardie, M. (1987), GRIP — a High-Performance Architecture for Parallel Graph Reduction, *in* FPCA'87 — Conference on Functional Programming Languages and Computer Architecture, LNCS 274, Springer-Verlag, Portland, OR, USA, September 14–16, pp. 98–112. (pp. 35, 40, 43, 46, 52, 58)

Peyton Jones, S., Gordon, A. & Finne, S. (1996), Concurrent Haskell, *in* POPL'96 — Symposium on Principles of Programming Languages, ACM Press, St Petersburg, FL, USA, January 21–24, pp. 295–308. (p 56)
**URL:** `http://www.dcs.gla.ac.uk/fp/authors/Simon_Peyton_Jones/-concurrent-haskell.ps.gz`

Peyton Jones, S., Hall, C., Hammond, K., Partain, W. & Wadler, P. (1993), The Glasgow Haskell Compiler: a Technical Overview, *in* Joint Framework for Information Technology Technical Conference, Keele, UK, pp. 249–257. (p 9)
**URL:** `http://www.dcs.gla.ac.uk/fp/papers/grasp-jfit.ps.Z`

Peyton Jones, S. & Wadler, P. (1993), Imperative Functional Programming, *in* POPL'93 — Symposium on Principles of Programming Languages, ACM Press, Charlotte, NC, USA, pp. 71–84. (p 70)
**URL:** `http://cm.bell-labs.com/cm/cs/who/wadler/papers/imperative/-imperative.ps.gz`

Peyton Jones, S. (1987), *The Implementation of Functional Programming Languages*, Prentice-Hall. (pp. 27, 29)

Peyton Jones, S. (1989), Parallel Implementations of Functional Programming Languages, *The Computer Journal* **32**(2), 175–186. (p 29)

Peyton Jones, S. (1992), Implementing Lazy Functional Languages on Stock Hardware: the Spineless Tagless G-machine, *Journal of Functional Programming* **2**(2), 127–202. (pp. 29, 51, 55)
**URL:** `ftp://ftp.dcs.gla.ac.uk/pub/glasgow-fp/papers/-spineless-tagless-gmachine.ps.Z`

Peyton Jones, S. (1996), Compiling Haskell by Program Transformation: a Report from the Trenches, *in* ESOP'96 — European Symposium on Programming, LNCS 1058, Springer-Verlag, Linköping, Sweden, April 22–24, pp. 18–44. (p 9)
**URL:** `http://www.dcs.gla.ac.uk/fp/authors/Simon_Peyton_Jones/-comp-by-trans.ps.gz`

Plainfossé, D. & Shapiro, M. (1995), A Survey of Distributed Garbage Collection Techniques, *in* IWMM'95 — International Workshop on Memory Management, Kinross, Scotland, UK. (p 42)
**URL:** `ftp://ftp.inria.fr/INRIA/Projects/SOR/SDGC_iwmm95.ps.gz`

Plotkin, G. (1981), A Structural Approach to Operational Semantics, Technical Report DAIMI-FN-19, Computer Science Department, Aarhus University. (p 194)

Pugh, W. (1992), The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis, *Communications of the ACM* **8**, 102–114.

(pp. 200, 201)
**URL:** `ftp://ftp.cs.umd.edu/pub/omega/techReports/non-TRs/omega/-omega.ps.Z`

Rabhi, F. & Manson, G. (1990), Using Complexity Functions to Control Parallelism in Functional Programs, Technical Report CS-90-1, Department of Computer Science, University of Sheffield. (p 182)

Rabhi, F. (1992), Run-Time Control of the Granularity in Functional Languages, *in* European Workshop on Parallel Computing, IOS Press, Barcelona, Spain, pp. 356–359. (p 182)

Rabhi, F. (1995), A Parallel Programming Methodology Based on Paradigms, *in* Transputer and Occam Developments, IOS Press, pp. 239–252. (pp. 145, 184)
**URL:** `http://www.enc.hull.ac.uk/~far/methodology.ps.gz`

Rangaswami, R. (1996), A Cost Analysis for a Higher-order Parallel Programming Model, PhD thesis, Department of Computer Science, University of Edinburgh. (p 223)
**URL:** `http://www.dcs.ed.ac.uk/home/ror/THESIS/thesis.ps.gz`

Reistad, B. & Gifford, D. (1994), Static Dependent Costs for Estimating Execution Time, *in* LFP'94 — Conference on Lisp and Functional Programming, ACM Press, Orlando, FL, USA, June 27–29, pp. 65–78. (pp. 12, 188, 193, 199, 200, 201, 206, 207, 214, 224, 228, 231, 236)
**URL:** `http://www-psrg.lcs.mit.edu/ftpdir/pub/reistad/lfp94.ps`

Rice (1993), *High Performance Fortran Language Specification*, 1.1 edition. (p 6)
**URL:** `http://www.erc.msstate.edu/hpff/hpf-report-ps/hpf-v11.ps`

Robinson, J. (1965), A Machine Oriented Logic Based on the Resolution Principle, *Communications of the ACM* **12**(1), 23–41. (p 204)

Roe, P. (1991), Parallel Programming Using Functional Languages, PhD thesis, Department of Computing Science, University of Glasgow. (pp. 41, 54)

Rosendahl, M. (1986), Automatic Program Analysis, Master's thesis, DIKU, University of Copenhagen. (pp. 188, 201, 225, 228)

Rosendahl, M. (1989), Automatic Complexity Analysis, *in* FPCA'89 — Conference on Functional Programming Languages and Computer Architecture, ACM Press, Imperial College, London, UK, September 11–13, pp. 144–156. (pp. 224, 225)
**URL:** `ftp://ftp.diku.dk/diku/semantics/papers/D-36.dvi.Z`

Ruggiero, C. & Sargeant, J. (1987), Control of Parallelism in the Manchester Dataflow Machine, *in* FPCA'87 — Conference on Functional Programming Languages and Computer Architecture, LNCS 274, Springer-Verlag, Portland, OR, USA, September 14–16, pp. 1–15. (pp. 50, 181)

Runciman, C. & Wakeling, D. (1993), Profiling Parallel Functional Computations (without Parallel Machines), *in* Glasgow Workshop on Functional Programming, Workshops in Computing, Springer-Verlag, Ayr, Scotland, UK, July 5–7, pp. 203–214. (pp. 54, 76, 80)

Runciman, C. & Wakeling, D. (1995), *Applications of Functional Programming*, UCL Press. (pp. 69, 152)

Rushall, D. (1995), Task Exposure in the Parallel Implementation of Functional Programming Languages, PhD thesis, Department of Computer Science, University of Manchester. (pp. 21, 34, 35, 179)

Sands, D. (1990*a*), Calculi for Time Analysis of Functional Programs, PhD thesis, Imperial College, University of London. (p 227)

Sands, D. (1990*b*), Complexity Analysis for a Lazy Higher-Order Language, *in* ESOP'90 — European Symposium on Programming, LNCS 432, Springer-Verlag, Copenhagen, Denmark, May 15–18, pp. 361–376. (p 240)

Sansom, P. & Peyton Jones, S. (1995), Time and Space Profiling for Non-Strict Higher-Order Functional Languages, *in* POPL'95 — Symposium on Principles of Programming Languages, ACM Press, San Francisco, CA, USA. (pp. 68, 143, 238)
**URL:** `ftp://ftp.dcs.glasgow.ac.uk/pub/glasgow-fp/papers/profiling.ps.Z`

Santos, A. (1995), Compilation by Transformation in Non-Strict Functional Languages, PhD thesis, Department of Computing Science, University of Glasgow. (p 78)

Sargeant, J. (1991), Use of Lazy Task Creation in Dynamic Computations, Internal EDS report EDS.RD.3I.M017, Department of Computer Science, University of Manchester. (p 177)

Sargeant, J. (1993), Improving Compilation of Implicit Parallel Programs by Using Runtime Information, *in* Workshop on the Compilation of Symbolic Languages for Parallel Computers, Technical Report ANL-91/34, Argonne National Laboratory, pp. 129–148. (p 183)

Sarkar, V. & Hennessy, J. (1986), Partitioning Parallel Programs for Macro-Dataflow, *in* LFP'86 — Conference on Lisp and Functional Programming, ACM Press, Cambridge, MA, USA, August 4–6, pp. 202–211. (p 30)

Sarkar, V. (1989), *Partitioning and Scheduling Parallel Programs for Multiprocessors*, Research Monographs in Parallel and Distributed Computing, MIT Press. (p 159)

Schreiner, W. (1993), Parallel Functional Programming (An Annotated Bibliography), Technical Report 93-24, RISC-Linz, Johannes Kepler University, Linz. (p 27)
**URL:** `ftp://ftp.risc.uni-linz.ac.at/pub/techreports/1993/93-24.ps.gz`

Seward, J. (1995), Abstract Interpretation of Functional Languages: A Quantitative Assessment, PhD thesis, Department of Computer Science, University of Manchester. (p 190)

Shaw, A., Arvind & Johnson, R. (1996), Performance Tuning Scientific Codes for Dataflow Execution, *in* PACT'96 — International Conference on Parallel Architecture and Compilation Techniques. (pp. 8, 153, 182)
**URL:** `ftp://csg-ftp.lcs.mit.edu/pub/papers/csgmemo/memo-381.ps`

Shaw, A. (1998), Impala Application Suite, WWW page. (p 155)
**URL:** `http://www.csg.lcs.mit.edu:8001/impala/`

Shimada, T., Hiraki, K., Nishida, K. & Sekigushi, S. (1986), Evaluation of a Prototype Data Flow Processor of the Sigma-1 for Scientific Computations, *in* ISCA'86 — International Symposium on Computer Architecture, IEEE Computer Society, Tokyo, Japan, June 2–5, pp. 226–234. (p 30)

Skillicorn, D. & Cai, W. (1993), A Cost Calculus for Parallel Functional Programming, Technical Report, Queens University, Kingston, Canada. (pp. 209, 223)
**URL:** `ftp://ftp.qucis.queensu.ca/pub/skill/costcalculusII.ps.Z`

Smirni, E., Merlo, A., Tessera, D., Haring, G. & Kotsis, G. (1995), Modelling Speedup of SPMD Applications on the Intel Paragon: a Case Study, *in* HPCN'95 — High Performance Computing and Networking, LNCS 919, Springer-Verlag, Milan, Italy, pp. 94–101. (p 17)

Sodan, A. & Bock, H. (1995), Extracting Characteristics from Functional Programs for Mapping to Massively Parallel Machines, *in* HPFC'95 — High Performance Functional Computing, Denver, CO, USA, April 10–12, pp. 134–148. (p 183)
**URL:** `ftp://sisal.llnl.gov/pub/hpfc/papers95/paper14.ps`

Sur, S. & Böhm, W. (1994*a*), Analysis of Non-Strict Functional Implementations of the Dongarra-Sorensen Eigensolver, *in* ICS'94 — International Conference on Supercomputing, Manchester, UK. (p 153)
**URL:** `http://www.cs.colostate.edu/~dataflow/papers/ics94b.ps.gz`

Sur, S. & Böhm, W. (1994*b*), Functional, I-Structure, and M-Structure Implementations of NAS Benchmark FT, *in* PACT'94 — International Conference on Parallel Architecture and Compilation Techniques, Montreal, Canada, pp. 47–56. (p 154)
**URL:** `http://www.cs.colostate.edu/~dataflow/papers/pact94b.ps.gz`

Talpin, J.-P. & Jouvelot, P. (1992), Polymorphic Type, Region and Effect Inference, *Journal of Functional Programming* **2**(3), 245–271. (pp. 207, 214, 216)

Taura, K., Matsuoka, S. & Yonezawa, A. (1994), StackThreads: An Abstract Machine for Scheduling Fine-Grain Threads on Stock Cpus, *in* Workshop on Theory and Practice of Parallel Programming, LNCS 907, Springer-Verlag, pp. 121–136. (p 177)
**URL:** `ftp://ftp.yl.is.s.u-tokyo.ac.jp/pub/papers/-jspp94-stackthreads-a4.ps.gz`

Tick, E. & Zhong, X. (1993), A Compile-Time Granularity Analysis Algorithm and its Performance Evaluation, *New Generation Computing* **11**(3-4), 271–295. (p 227)

Tofte, M. (1988), Operational Semantics and Polymorphic Type Inference, PhD thesis, University of Edinburgh. (p 194)
**URL:** `http://www.diku.dk/users/tofte/publ/thesis-part1and2.ps`

Traub, K., Culler, D. & Schauser, K. (1992), Global Analysis for Partitioning Non-Strict Programs into Sequential Threads, *in* LFP'92 — Conference on LISP and Functional Programming, San Francisco, CA, USA. (p 31)

Trinder, P., Hammond, K., Loidl, H.-W. & Peyton Jones, S. (1998), Algorithm + Strategy = Parallelism, *Journal of Functional Programming* **8**(1). (pp. xviii, 12, 91, 182, 236)
**URL:** `http://www.dcs.gla.ac.uk/~hwloidl/publications/strategies.ps.gz`

Trinder, P., Hammond, K., Loidl, H.-W., Peyton Jones, S. & Wu, J. (1998), Go-faster Haskell; Or: Data-intensive Programming in Parallel Haskell, *in* ICFP'98 — International Conference on Functional Programming, Baltimore, MD, USA, September 27–29. Submitted for publication. (pp. 79, 143, 155)

Trinder, P., Hammond, K., Mattson Jr., J., Partridge, A. & Peyton Jones, S. (1996), GUM: a Portable Parallel Implementation of Haskell, *in* PLDI'96 — Programming Languages Design and Implementation, Philadelphia, PA, USA, pp. 79–88. (pp. 35, 36, 41, 43, 46, 48, 55, 65, 97, 143)
**URL:** `ftp://ftp.dcs.glasgow.ac.uk/pub/glasgow-fp/authors/Philip_Trinder/-gumFinal.ps.Z`

Turner, D. (1979), A New Implementation Technique for Applicative Languages, *Software – Practice and Experience* **9**, 31–49. (p 27)

van Eekelen, M. & Plasmeijer, M. (1993), Process Annotations and Process Types, *in* Term Graph Rewriting — Theory and Practice, John Wiley & Sons, pp. 347–362. (p 183)
**URL:** `ftp://ftp.cs.kun.nl/pub/CSI/SoftwEng.FunctLang/papers/-eekm93-Processes&Types.ps.gz`

van Groningen, J. (1992), Some Implementation Aspects of Concurrent Clean on Distributed Memory Architectures, *in* IFL'92 — International Workshop on the Parallel Implementation of Functional Languages, RWTH Aachen, Germany, September 28–30. (pp. 37, 54)

Wadler, P. & Hughes, R. (1987), Projections for Strictness Analysis, *in* FPCA'87 — Conference on Functional Programming Languages and Computer Architecture, LNCS 274, Springer-Verlag, Portland, OR, USA, September 14–16, pp. 385–407. (p 226)
**URL:** `http://www.dcs.glasgow.ac.uk/~wadler/papers/strictproject/-context.ps`

Wadler, P. (1988), Strictness Analysis Aids Time Analysis, *in* POPL'88 — Symposium on Principles of Programming Languages, ACM Press, San Diego, CA, USA. (p 227)
**URL:** `ftp://ftp.dcs.gla.ac.uk/pub/glasgow-fp/authors/Philip_Wadler/-stricttime.dvi`

Wadler, P. (1998), The Marriage of Effects and Monads, Draft paper. (p 224)
**URL:** `http://cm.bell-labs.com/cm/cs/who/wadler/papers/effects/-effects.ps.gz`

Wadsworth, C. (1971), Semantics and Pragmatics of the Lambda Calculus, PhD thesis, University of Oxford. (p 27)

Warren, D. (1983), An Abstract Prolog Instruction Set, Technical report 309, SRI International. (p 44)

Watson, I. (1988), A LAGER Execution Mechanism and Store Model, Internal Report Version 2, Department of Computer Science, University of Manchester. (pp. 39, 50, 163)

Watson, I. (1989), Simulation of a Physical EDS Machine Architecture, Technical report, Department of Computer Science, University of Manchester. (p 54)

Watson, I. (1990), C-LAGER Definition, Internal Document EDS.UD.3I.M0004, University of Manchester. (p 177)

Wegbreit, B. (1975), Mechanical Program Analysis, *Communications of the ACM* **18**(9), 528–539. (p 222)

Winstanley, N. (1997), A Type-Sensitive Preprocessor for Haskell, *in* Glasgow Workshop on Functional Programming, Ullapool, Scotland, UK, September 15–17. Submitted for publication. (p 122)
**URL:** `http://www.dcs.gla.ac.uk/~nww/Papers/GlaFP.ps.Z`

Wolfram, S. (1988), *Mathematica — A System for Doing Mathematics by Computer*, Addison-Wesley. (p 213)

Wu, J. & Harbird, L. (1996), A Functional Database System for Road Accident Analysis, *Advances in Engineering Software* **26**(1), 29–43. (p 79)

Zimmermann, W. (1990), Automatische Komplexitätsanalyse von funktionalen Programmen (Automatic Complexity Analysis of Functional Programs) (in German), PhD thesis, University of Karlsruhe. (p 223)