Engineering Parallel Symbolic Programs in GpH

Hans-Wolfgang Loidl, Philip W. Trinder,
Kevin Hammond, Sahalu B. Junaidu,
Richard G. Morgan, Simon L. Peyton Jones

October 5, 1998

Abstract

We investigate the claim that functional languages offer low-cost parallelism in the context of symbolic programs on modest parallel architectures. In our investigation we present the first comparative study of the construction of large applications in a parallel functional language, in our case in Glasgow Parallel Haskell (GpH). The applications cover a range of application areas, use several parallel programming paradigms, and are measured on two very different parallel architectures.

On the applications level the most significant result is that we are able to achieve modest wall-clock speedups (between factors of 2 and 10) over the optimised sequential versions for all but one of the programs. Speedups are obtained even for programs that were not written with the intention of being parallelised. These gains are achieved with a relatively small programmer-effort. One reason for the relative ease of parallelisation is the use of evaluation strategies, a new parallel programming technique that separates the algorithm from the coordination of parallel behaviour.

On the language level we show that the combination of lazy and parallel evaluation is useful for achieving a high level of abstraction. In particular we can describe top-level parallelism, and also preserve module abstraction by describing parallelism over the data structures provided at the module interface ("data-oriented parallelism"). Furthermore, we find that the determinism of the language is helpful, as is the largely-implicit nature of parallelism in GpH.

1 Introduction

Parallelism without pain is perpetually promised — but seldom delivered. For applications where the parallelism is well-structured, well-understood techniques such as SPMD now deliver good performance [SMT+95]. But for richly-structured symbolic applications, such as compilers and natural-language processing, the jury is still out. Such applications are characterised as follows.

- The computation is largely symbolic, rather than numerical, e.g. with arbitrary precision integers rather than floating point numbers.
- The data structures are complex, e.g. richly connected trees or graphs, rather than arrays.
- The algorithm supports modest, rather than massive, parallelism.
- Parallelism arises from several sources, often nested within one another.
- Thread granularities are not statically predictable.

The literature on parallel applications of this sort is sparse, and good results seem to demand an unreasonable investment of effort, except in particularly well-studied niches, such as parallel discrete event simulation and computer algebra [JSC96].

Functional programming languages have long held out the possibility of addressing parallel symbolic applications. On the one hand, their automatic storage allocation, polymorphic typing, and rich data structures, makes them well suited to symbolic applications. On the other hand, their expression-oriented style exposes much potential parallelism.

Despite this promise, real parallel implementations have been slow in coming. By a "real" implementation we mean one that (a) delivers wall-clock speedups over the best sequential compiler for the same language, and (b) is robust enough to handle multi-thousand-line application programs. The engineering challenge of developing a real implementation in this sense is considerable. Hammond [Ham94] provides a good overview of work in this area, and Section 7 discusses related work on applications.

We have, however, developed a real implementation of the functional language Haskell [PHA+97], described in [THM+96]. Using it we have begun to write substantial parallel applications, and to develop systematic ways of doing so. In this paper we describe our experiences of parallelising a set of five parallel applications of varying size. Three are really warm-up exercises, serving to set the scene. The last two, a compiler for Haskell, and a natural-language processing system are substantial: 5,000 and 47,000 lines of Haskell respectively. Together, these applications cover a range of

- application areas;
- parallel programming paradigms;
- parallel computer systems.

So, based on this experience, what is the verdict? Our conclusions are these:

- With a modest investment of effort, it is possible to extract modest levels of parallelism (a factor of 2–10), and wall-clock speedup, for complex symbolic applications that were originally written without parallelism in mind (Section 5.1). It can be difficult to extract much more parallelism than this without substantial rewriting.

  Viewed from the massively-parallel computing standpoint, this looks disappointing. Viewed from the position of a compiler writer used to considering a 20% improvement as a huge win, it looks exciting. "Low pain, moderate gain" is our motto. Because this speedup is achieved with only minor changes in the code, merely exposing parallelism rather than controlling it in detail, this style of parallelism should be of interest for non-specialists in parallel programming.

- Some of the long-time claims of the functional community do hold good. In particular, determinism is an enormous boon. Once a program works on a uni-processor, then it also works on a multi-processor, and always delivers the same results. There are no race hazards, core dumps, and un-repeatable errors. However, the usual problems and advantages of different resource usage in a multi-processor setting remain, as illustrated in Section 4.5.

- We have found a way to cleanly separate the *algorithm* that computes the result from the *evaluation strategy* that governs its parallel behaviour. Evaluation strategies are the topic of another paper [THLP98], and are introduced in Section 2.1.

  Interestingly, *lazy evaluation* plays an essential role in supporting this modular program decomposition. (Lazy evaluation means that a component of a data structure is only evaluated when its value is needed.) This result directly contradicts the folk-lore that laziness and parallelism are in conflict [TG95, Ken94]. In short, lazy evaluation allows to define parallelism over a data structure produced by a function without breaking the abstraction of the function. This data-oriented form of parallel programming encourages a modular design where sequential functions can be reused and parallelism is defined when composing several functions.

- Our techniques support a variety of parallel programming paradigms, including farms, pipelines, divide-and-conquer, and data parallelism. Since some of our applications involve several different forms of parallelism, it is helpful that our programming framework is not biased towards one particular paradigm. Several of the programs nest one paradigm within another, furthermore we exploit the facility to nest paradigms to an arbitrary depth.

These are general remarks. The distinctive contribution of this paper is that we justify them in detail, based on experience of substantial applications covering a range of application areas.

Parallel functional programming is no panacea. Writing parallel algorithms is still hard. For applications that demand very high utilisation of an expensive massively-parallel machine the programmer might well be better off with existing approaches. However, in an age where every desktop machine will soon be a multi-processor, and where under-used networks of workstations abound, a way to extract modest speedups for a modest investment of effort is a welcome and encouraging development.

The structure of the paper is as follows. After discussing the programming language in Section 2 and environment in Section 3, we describe the applications themselves in Section 4. In the rest of the paper we then try to abstract the lessons we learned from that experience in Sections 5 and 6. We include a substantial survey of the field in Section 7, before concluding with Section 8.

2 GPH — A Parallel Functional Language

The essence of the problem facing the parallel programmer is that, in addition to specifying *what* value the program should compute, explicitly-parallel programs must also specify *how* the machine should organise the computation. There are many aspects to the parallel execution of a program: threads are created, execute on a processor, transfer data to and from remote processors, and synchronise with other threads, etc. Managing all of these aspects on top of constructing a correct and efficient algorithm is what makes *explicit* parallel programming so hard. The diametrically opposing approach is to rely solely on the compiler and runtime system to manage the parallel execution without any programmer input. Unfortunately, this purely *implicit* approach is not yet fruitful for the large-scale functional programs we are interested in.

The approach used in GPH is intermediate between purely implicit and purely explicit approaches. The runtime system manages most of the parallel execution, only requiring the programmer to indicate those values that might usefully be evaluated by parallel threads and, since our basic execution model is a lazy one, perhaps also the extent to which those values should be evaluated. We term these programmer-specified aspects the program's *dynamic behaviour*.

Parallelism is introduced in GPH by the `par` combinator, which takes two arguments that are to be evaluated in parallel. The expression `p `par` e` (here we use Haskell's infix operator notation) has the same value as `e`, and is not strict in its first argument, i.e. $\perp$ `` `par` e`` has the value of `e`. Its dynamic behaviour is to indicate that `p` could be evaluated by a new parallel thread, with the parent thread continuing evaluation of `e`. We say that `p` has been *sparked*, and a thread may subsequently be created to evaluate it if a processor becomes idle. Since the thread is not necessarily created, `p` is similar to a *lazy future* [MKH91].

Since control of sequencing can be important in a parallel language [Roe91], we introduce a sequential composition operator, `seq`. If `e1` is not $\perp$, the expression `e1 `seq` e2` also has the value of `e2`; otherwise it is $\perp$. The corresponding dynamic behaviour is to evaluate `e1` to weak head normal form (WHNF) before returning `e2`.

This section gives an abridged introduction to our parallel programming technique called evaluation strategies. We focus on the language features necessary to achieve the basic functionality and highlight the advantages of this parallel programming technique. A complete description and discussion of evaluation strategies can be found in [THLP98].

2.1 Evaluation Strategies

Even with the simple parallel programming model provided by `par` and `seq` we find that more and more code is inserted in order to obtain better parallel performance. In realistic programs the algorithm can become entirely obscured by the dynamic-behaviour code.

Evaluation strategies use lazy higher-order functions to separate the two concerns of specifying the algorithm and specifying the program's dynamic behaviour. A function definition is split into two parts, the *algorithm* and the *strategy*, with values defined in the former being manipulated in the latter. The algorithmic code is consequently uncluttered by details relating only to the dynamic behaviour. In fact the driving philosophy behind evaluation strategies is that *it should be possible to understand the semantics of a function without considering its dynamic behaviour.*

A 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 empty tuple `()`.

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

2.1.1 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 both built-in and user-defined types, but not over function types or any type incorporating a function type — 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
```

2.1.2 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.

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

A parallel map is an example of data-oriented parallelism, and is used in several of the programs. The `parMap` function defined below applies its function argument to every element of a list in parallel. 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
```

As an alternative to such a `using`-based design of parallel code we have also introduced a new construct, `$||`, called strategic function application. As an extension to the standard function application, `$`, in Haskell, the construct `f $|| s $ x` applies the strategy `s` to the argument `x` in parallel with applying the function `f` to `x`. This construct is especially useful for defining data-oriented parallelism over complex data-structures. This is due to the typical design of functional programs as compositions of small, flexible sub-functions [Hug89]. Compared to the above `parMap` function this new construct makes it possible to define data-oriented parallelism without changing the definition of `map` itself. For example the expression `g $ parMap rnf f xs` can also be written as

```
g $|| parList rnf $ map f xs
```

In the latter expression the strategy is separated from the algorithmic code and the sequential sub-functions are unchanged, thus describing parallelism on a higher level in the program. Variants of this idea are sequential strategic function application, `$|`, which adds a synchronisation barrier and thus is useful for defining pipelines, and strategic function composition in a parallel, `.||`, and a sequential version, `.|`, respectively.

2.2 Summary

The prime motivation in the design of evaluation strategies has been the separation of algorithmic and behavioural code. This separation will be discussed together with the applications in Section 4. A comparison of pre-strategy with strategic code, as given in [Loi97], shows that such a separation aids the performance tuning process of parallel programs and enables the programmer to experiment with several parallel versions of the code.

Because evaluation strategies are written using the same language as the algorithm, they have additional desirable properties. Strategies are powerful: simpler strategies can be composed, or passed as arguments to form more elaborate strategies. Strategies are extensible: indeed in the parallelisation of several of the programs in Section 4 we have defined new application-specific strategies. Strategies can be defined over all types in the language, and offer some level of type safety because the normal type system applies to strategic code. Strategies have a clear semantics, which is precisely that used by the algorithmic language.

3 Parallel Programming Environment

GpH programs are developed with an integrated suite of software tools, based on the Glasgow Haskell Compiler, GHC [Pey96]. Guidelines for the use of these tools are given in the following subsection. The suite includes both a development environment and dynamic analysis tools, as outlined below. A more detailed discussion of the parallel programming environment is given in [TBD+98].

- The Hugs *interpreter*, for fast development, experimentation and debugging of sequential code. Being an interpreter, Hugs offers fast turn-around time for code changes and an interactive development environment. This comes at the expense of higher execution time compared to GHC. In an ongoing project these two components, Hugs and GHC, are combined into a single environment, which we could reuse in our parallel programming environment.
- The GHC *compiler and sequential runtime system* for fast execution of sequential code. GHC is a state-of-the-art optimising compiler for Haskell. Thus our programs do not sacrifice sequential performance in order to achieve good parallelism. Another advantage of this embedding of GpH into Haskell is, that all future work on sequential program analysis and optimisation can be automatically reused in the parallel system. Most importantly, the parallel program has the same semantics as its sequential counterpart.
- The GHC *compiler and* GUM *parallel runtime system* for parallel execution on multiprocessors. GUM is efficient, robust and portable: being available on both shared- and distributed-memory architectures, including the Sun SPARCServer shared-memory multiprocessor and both a CM5 [Dav96] and networks of Sun and Alpha workstations. An IBM SP2 port is nearing completion. We discuss the architecture-independent aspect of our parallel system in [TBD+98]. GUM is freely available and has users and developers worldwide [THM+96].

The suite also has a number of analysis tools, most of them dynamic analysers, or profilers. Those used to construct the programs in Section 4 are as follows.

- *Sequential time and space profilers* are supplied with GHC [SP95]. They have proven indispensable in tuning large Haskell programs such as GHC itself.
- *The* GranSim *parameterisable parallel simulator* [HLP95, Loi98] is closely integrated with the GUM runtime system giving accurate results. It is parameterisable to emulate different target architectures, including an idealised machine, and provides a suite of visualisation tools to view aspects of the parallel execution of the program. The GUM runtime system produces a subset of the GranSim profile data and so can produce some of the profiles.

We are currently working on the development of a parallel profiler, which enables the programmer to connect points in an execution profile with statements in the source code. Initial results with GranCC, the prototype of a parallel profiler obtained by merging GranSim and sequential cost center profiling, achieved promising results and helped in the parallelisation of Naira [HLT97]. Currently, this approach is more tightly integrated into the parallel runtime system, and extended by tracking the evaluation history of parallel threads.

3.1 Parallelisation Guidelines

From our experiences engineering GpH programs we have developed some guidelines for constructing large non-strict functional programs. The guidelines are discussed in detail in [LT97, THLP98].

1. **Sequential implementation.** Start with a correct implementation of an inherently-parallel algorithm.
2. **Parallelise and tune**.

- *Seek top-level parallelism.* Often a program will operate over independent data items, or the program may have a pipeline structure.
- *Time Profile* the sequential application to discover the "big eaters", i.e. the computationally intensive pipeline stages.
- *Parallelise Big Eaters* using evaluation strategies.
- *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 isn't parallel on an idealised machine it won't be on a real machine.
- *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.

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

4 Parallel Programs

4.1 Introduction

This sections outlines five GpH programs, that cover a range of applications domains. The Alpha-Beta search is an AI search application; Accident Blackspots is a data-intensive application; LinSolv is a symbolic computation application; Naira is a compiler, and Lolita is a natural language-processor. Detailed descriptions of these programs have already been published in separate papers. Here we focus on common aspects of the programs and of the parallelisation process.

All of the programs except Alpha-Beta solve real problems with real data, although LinSolv should be viewed as a component of a larger system. The Alpha-Beta search program is included first because it is simple, and illustrates our approach.

The programs manipulate symbolic, rather than numerical data, using complex data structures, e.g. the forests of SGML trees found in Lolita, or arbitrary precision integers rather than floating point numbers in LinSolv.

None of the programs have a regular parallel structure. A typical program has a number of stages, and these can be linked in a pipeline and each stage uses a different parallel paradigm, e.g. data-parallel or divide-and-conquer. Some programs, like Naira, exhibit even deeper levels of nested parallelism. Because of this complex parallelism, neither the number of threads nor the granularity of the threads can be determined statically.

4.2 Alpha-Beta Search

4.2.1 Program Description

The Alpha-Beta search algorithm is typical of artificial intelligence applications. It is mainly used for game-playing programs to find the best next move. The sequential version of the algorithm presented here has been developed by John Hughes [Hug89] in order to demonstrate the strengths of lazy functional languages. Most notably, this algorithm relies on laziness to improve the efficiency of the naive sequential algorithm by pruning the search tree based on intermediate results. Therefore, the parallel version has to retain the laziness expressed in the sequential algorithm in order to avoid redundant work. In this section we parallelise this lazy functional algorithm and study the parallel runtime behaviour. We investigate the use of strategies to develop an efficient parallel algorithm without sacrificing the advantages of the original lazy algorithm. A more detailed discussion of two variants of this parallel algorithm is given in [LT97].

The Alpha-Beta algorithm examines the possible next moves and picks the best move for the player, assuming that the opponent picks the worst move for the player. The result is either

```
bestMove depth p f g = last                 .|| rwhnf $ -- list of approx
                       (mise f g)           .|| rwhnf $ -- cropped eval tree
                       cropTree             .|| rwhnf $ -- static eval tree
                       (mapTree (static p)) .|| rwhnf $ -- pruned search tree
                       (prune depth)        .|| rwhnf $ -- full search tree
                       repTree (newPositions p)
                               (newPositions (opposite p))
```

Figure 1: Parallel pipeline structure of choosing the best next move

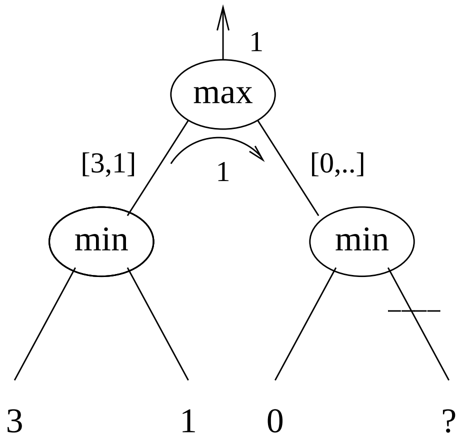

Figure 2: Pruning subtrees in the optimised Alpha-Beta algorithm

the maximum (player's move) or the minimum (opponent's move) of the evaluations of all next positions. Following a typical functional programming style, this algorithm can be very naturally described as a sequence of function compositions performing the following tasks (see Figure 1 ignoring the bold face parts of the code):

1. Build a tree with positions as nodes and all possible next moves as subtrees. Since this tree is built lazily no restrictions to its size apply. The higher-order function `repTree` is used to repeatedly apply a `newPosition` function to the nodes in the tree, alternating between the functions for the two players.

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

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. 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` and `g` represent the combination functions for the two players and alternate when traversing the tree.

6. The last element in the list of approximations returned by the `mise` function is the final value of the evaluation.

One crucial optimisation of the algorithm outlined above is the pruning of subtrees inside the `mise` function based on intermediate results. Figure 2 shows an example of the pruning process realised via lazy evaluation. Based on the result of the left subtree, the overall result must be at least 1, the last element of the list of approximations. Propagating this information as an intermediate result into the right subtree, we can prune this whole subtree after finding a value smaller than 0: since a `minimum` function is used to combine the result, it will be *at most* 0, which is smaller than the value we already have. It is not necessary to evaluate the unknown value in the rightmost subtree at all.

This dynamic behaviour is encoded as follows. The 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 inside `mise`, `minleq`, has to test whether the opponent's move from a subtree, represented as a decreasing list, can be ignored. 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: $\min\ l \ \leq\ x \ \Leftrightarrow:$ `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 [Hug89].

4.2.2 Parallelisation

Pipeline Parallelism. Considering the structure of the algorithm as a composition of several functions, our initial attempt of parallelising this algorithm was to add pipeline parallelism to the top level structure of the code. This approach has the advantage of modifying only a small portion of the overall code and has proven successful in parallelising large programs such as Lolita (see Section 4.5). The code in Figure 1 uses the strategic function composition operator `.||` to define the parallelism and the evaluation degree on the arguments of the individual functions.

Alas, the data dependencies of the algorithm do not permit the use of aggressive strategies. Therefore, only the weakest evaluation degree, `rwhnf`, is used in every stage, amounting to a pipeline structure with extremely short stages. Most of the work has to be performed by the final stage, resulting in virtually no speed up at all.

Data Parallelism. More promising than the pipeline parallel version is a data parallel approach. Our goal is to evaluate all possible next moves in parallel. The only necessary change to achieve this form of data parallelism affects the `mise` function in Stage 5 of the algorithm, shown in Figure 3. This function has to combine the results of all subtrees into a result at the current node. The parallel version of this function is shown in Figure 3. The only difference to the sequential version is the use of the `parMap rnf` strategy to capture a data parallel dynamic behaviour of this function. Depending on whether it is the player's or the opponent's move, the binary function `max` or `min` is taken as argument and folded over the list of results from the subtrees. Note that the functions `f` and `g` change position in the recursive call to record the switch in turns.

```
-- This does simple minimaxing without pruning subtrees
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 3: Data parallel combination function in the Alpha-Beta search algorithm

Unfortunately, this naive use of data parallelism generates a lot of redundant work because no pruning of subtrees is performed any more. This is indicated by the use of `rnf`, which fully evaluates the individual subtrees. Detailed measurements of variants of this algorithm in [LT97] reveal that the performance of this parallel algorithm is even worse than that of a naive parallel algorithm that omits any pruning of subtrees. Although the version in Figure 3 generates a lot of parallelism, most of it is speculative and therefore potentially redundant.

Data Parallelism with Pruning. In order to control the degree of speculative parallelism in the algorithm we force the evaluation of only an initial segment in the list of possible next positions.

```
-- Parallel version of the pruning version
mise :: Player -> Player -> (Tree Evaluation) -> [Evaluation]
mise f g (Branch a []) = [a]
mise f g (Branch _ l) =  -- force the first n elems of the result list
  f ((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: Strategy for a pruning Alpha-Beta search with a static force length

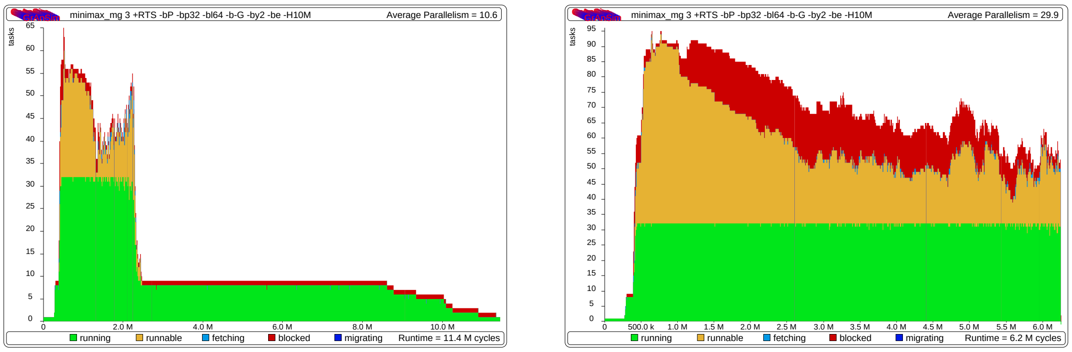

Figure 5: Data parallel versions with static force lengths of 0 and 4

We call the length of this segment the "force length". This parameter therefore represents a handle to tune the degree of speculative computation in the program. 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 code in Figure 4. The strategy in this code checks the value of the global variable `force_len` to decide how many possible next moves to evaluate. Since strategies are simply Haskell functions, the prelude function `take` for selecting an initial segment of a list can be used together with the corresponding function `drop`, which returns the rest of the list. Whereas `rnf` forces the evaluation of the whole list of approximations corresponding to a possible next move, `rwhnf` only evaluates the top level list cell, delaying any further computation.

Measurements. In order to demonstrate the effect of the force length parameter, Figure 5 compares the dynamic behaviour of Alpha-Beta search with a simple tic-tac-toe game, using two different force lengths. In all test runs we used a realistic GranSim setup modelling a tightly connected distributed memory machine with 32 processors, a latency of 64 machine cycles, and pre-fetching of data. In this case increasing the force length improves the average parallelism from 10.8 to 29.9, but the runtime only drops from 11.4 to 8.2 Mcycles. This indicates a high degree of speculative computation in the right hand graph.

More detailed measurements of this algorithm show that the largest speedup of 15.7 is obtained from a setup with a force length of 4. Of course, the optimal force length depends on the position to be analysed. For example if a winning position is found early on in the sequential algorithm only a poor speedup is achieved. However, with this additional parameter it is possible to control how much effort should be invested into potentially redundant work. Concrete runtimes and speedups for various variants of this algorithm and for different force lenghts are given in [LT97].

4.2.3 Discussion

The main interest in this algorithm lies in the interplay between lazy and parallel evaluation. Since the efficiency of this algorithm relies on the lazy traversal of the search tree, this laziness must be preserved in the parallel algorithm. Measurements in [LT97] show that in some cases a naive parallel algorithm without pruning is faster than a parallel algorithm with pruning, because in the latter the data parallel strategy destroys almost all possibilities of pruning.

On the other hand, Figure 5 shows that a conservative approach towards parallelism in the pruning version yields a very poor degree of parallelism. In order to improve the efficiency of the parallel version we had to introduce speculative parallelism into the program. We had to add an additional parameter to the key function in the program and we used strategies in order to express the speculative computation based on this parameter. Although the runtime-system of GranSim and GUM does not automatically kill threads that turn out to be unnecessary, thus running the risk of wasting resources, the resulting performance clearly exceeds the conservative parallel version. One difficulty in the tuning of the algorithm then lies in finding the right level of speculation in the program. In practice, this has to be chosen based on the concrete search problem that is implemented via an Alpha-Beta search algorithm.

4.3 Accident Blackspots

4.3.1 Program Description

The University of London Centre for Transport Studies wishes to analyse road traffic accident data. Given a set of police accident records (modified to preserve privacy) the task is to discover accident blackspots: locations where two or more accidents have occurred. A number of criteria can be used to determine whether two accident reports are for the same location. Two accidents may be at the same location if they occurred at the same junction number, at the same pair of roads, at the same grid reference, or within a small radius of each other. The radius is determined by the class of the roads, type of the junction etc. The problem is obviously data-intensive, and too complex for conventional database query languages like SQL.

Locating blackspots amounts to combining several partitions of a set into a single partition. For example if the partition on road pairs is `{{2,4,5},{3},{6,7}}` and on grid references is `{{2,5},{3},{4,6},{7}}`, the combined partition is `{{2,4,5,6,7},{3}}`. The problem of unioning disjoint sets, *union find*, has been much studied by algorithm designers as it has an interesting sequential complexity. For n union and m find operations, an algorithm with an amortised complexity of O(n + F(m,n)) can be given, where F is a very small function (the inverse of the Ackermann function) [Tar75]. These RAM algorithms are not directly applicable in our application because not all of a large data set may be randomly accessed in memory. We have adopted an index-, or tree-, based solution with complexity O(n log n) if n is the number of elements in the sets. The motivation for this choice is that for very large data sets not all of the tree need be memory resident at any time.

Sequential Implementations. The application was originally written at the Centre for Transport Studies [WH96] in PFL and has subsequently been rewritten in Haskell. PFL is an interpreted functional language [PS93], designed specifically to handle large deductive databases. Unusually for a functional language, PFL provides a uniform persistent framework for both data and program. The PFL program uses selectors, a special bulk-data manipulating construct, and hence an algorithm that is slightly different from that used in the Haskell program. It comprises approximately 500 lines.

The Haskell implementation constructs a binary *sameSite* relation containing an element for each pair of accidents that match under one of the four conditions. The combined partition is formed by repeatedly finding all of the accidents reachable in *sameSite* from a given accident. The program has four major phases: reading and parsing the file of accidents; building indices over the accident data; constructing sameSite, and indices over sameSite; forming the partition. The program is a 300-line module, together with 3 library modules totalling 1300 lines.

Table 1: Idealised simulation

Parallel Variant	Work (MCycles)	Average Parallelism	Run Time (MCycles)
Pipeline only	327	1.2	273
Par. Pipeline Stages	327	2.8	124
Par. Pipeline Stages & preconstructed Ixs	304	3.5	87
Geographically Partitioned (Tiled)	389	3.7	105

Table 2: Realistic SPARCserver simulation

Parallel Variant	Work (MCycles)	Average Parallelism	Run Time (MCycles)
Par. Pipeline Stages & preconstructed Ixs	393	2.3	171
Geographically Partitioned (Tiled)	394	3.7	105

The original data set comprises 7310 accident reports, and the programs discover 1229 multiple-accident sites where a total of 5450 accident occur. The programs are run on similar, but not identical, workstations: PFL on a Sun ELC, and Haskell on a Sun Sparc Classic. The runtimes of the programs are as follows, PFL: 1105 seconds, Haskell: 123 seconds. The faster execution of the Haskell program is attributed to it being both compiled and highly optimised, where PFL is an interpreted research language. More measurements of the PFL and Haskell programs, together with a more detailed discussion can be found in [THLP98].

4.3.2 Parallelisation

Simulated Parallel Variants. Following the guidelines, we initially investigated the application's parallelism using an idealised simulation. Once adequate parallelism was obtained, we used a realistic simulation of our first 4-processor shared-memory target machine. Tables 1 and 2 report the results obtained from the simulators when just 1000 accidents are partitioned, runtimes and work are in units of 10^6 GRANSIM machine cycles.

Pipeline only. The first version simply converted the 4 phases of the program outlined in section 4.3.1 into a pipeline. The speedup of 1.2 is low because the pipeline is blocked by the trees passed between stages.

Parallel Pipeline Stages. The next version introduces parallelism within each pipeline stage using a variety of paradigms, as discussed below.

Parallel Pipeline Stages and Preconstructed Indices. Parallelism is further improved by merging the first two pipeline stages. That is, the indices on the accident data were constructed before the program is run, and the program reads the indices from a file rather than constructing them. The resulting parallelism is satisfactory on an idealised simulation of a 4-processor machine, but poor under a realistic simulation. The poor realistic results are due to the fine grain of parallelism and the volume of data being communicated.

Geographically Partitioned (Tiled). A very different, coarse-grained, parallel structure can be obtained by splitting the accident data into geographical areas. Each area, or *tile*, can

Table 3: Monolithic and tiled runtimes

Program Variant	Work (MCycles)	Average Parallelism	Run Time (MCycles)
Sequential Monolithic	498	1.0	498
Sequential Tiled	394	1.0	394
Parallel Tiles	394	3.7	105

be partitioned in parallel before aggregating the results, using this standard technique [MS95]. Accidents occurring near the edges of a tile must be treated specially. This approach is only feasible because every accident has a grid reference and we assume that accidents occurring more than 200m apart cannot be at the same site. Accidents occurring within 100m of the nominal edge between two tiles are duplicated in both tiles. Splitting the original data into 4 tiles results in a 4% increase in data volume. As a result of the duplicated border accidents, some multiple-accident sites may be discovered in more than one tile.

Breaking the data into tiles reduces the work required to form a partition as long as the the border is sufficiently smaller than the body of the tile. Less work is required because each accident is compared with fewer accidents: the trees constructed during the partition are smaller. Table 3 shows the runtimes for a sequential partition of the original (monolithic) set of accidents, a sequential partition of the data in 4 tiles, and a parallel partition of the 4 tiles. More formally, for the n accidents in the monolithic data, the algorithm is $O(n \log n)$, whereas if we assume that the borders are sufficiently small, then the tiled algorithm is $O(n \log n/4)$.

Parallel Machine Measurements. The program is measured on two very different machines, making use of the portability of the GUM runtime system. One is a shared-memory architecture and the other distributed-memory. The shared-memory machine is a Sun SPARCserver with 4 Sparc 10 processors and 256Mb of RAM. The machine is shared with other users, but measurements are performed when it is very lightly loaded. The distributed-memory machine is a network of up to 16 Sun 4/15 workstations each with 24Mb of RAM, and connected on a single ethernet segment. Both architectures use a shared file system, i.e. any PE can access any file. On the network of workstations the files are stored on a single file server and accessed via NFS.

Data. The original data set of 7310 accident reports occupies 0.3Mb and is too small to obtain good results on the parallel machines. For the purposes of this section, the data is replicated 6 times. The larger data set could be kept in larger tiles, or in more tiles of the same size, and the latter approach is taken for the following reasons. As shown in Section 4.3.2, as long as the tiles are large relative to the border area, many smaller tiles are more efficient than a few large tiles. Peak resource usage is reduced because if there is one tile per PE then all of the file reading occurs at the start of the program, inducing intense network traffic. With mulitple tiles per PE the file reading is spread through the program execution. Multiple tiles utilise the dynamic load management provided by GUM, demonstrating that the GpH program is independent both of the number of PEs and of the number and size of tiles. In contrast a small number of large tiles could be statically allocated to PEs. However it is a tedious task to maintain the allocation as the number of tiles and PEs change.

The replicated data occupies 1.8 Mb and is split into 40 tiles with two different sizes. There are 32 *small* tiles, each containing approximately 1000 accidents and occupying 37Kb, and 8 *large* tiles each containing approximately 2000 accidents and occupying 73Kb.

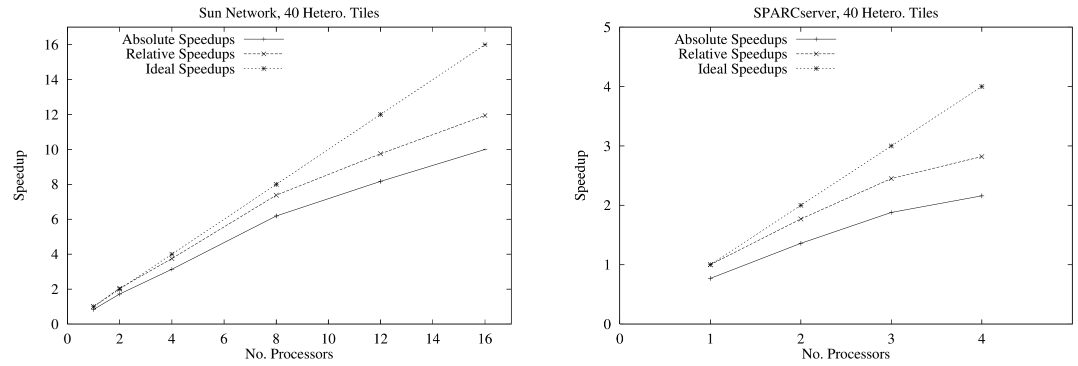

Figure 6: Speedups of Blackspots on heterogeneous tiles

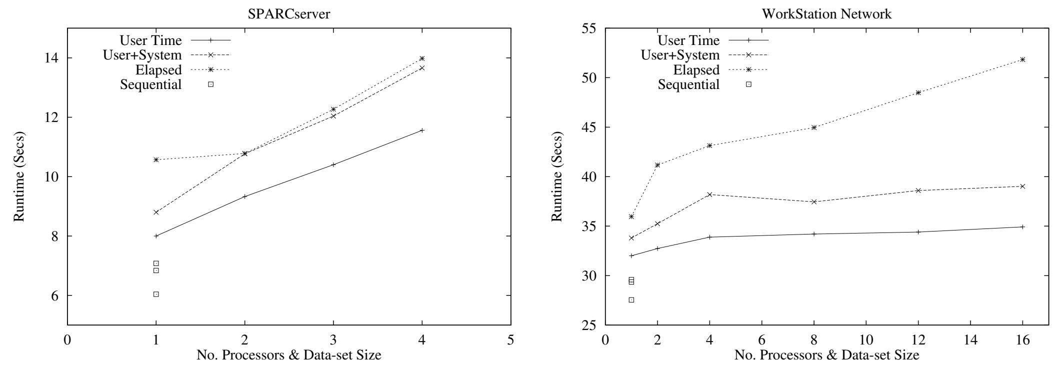

Figure 7: Application scaleup

Program. Only one change is required to the GRANSIM version of the program to enable it to run under GUM. GUM processes don't inherit file handles from the main thread, and hence to permit them to read files the program uses the 'unsafe' C-interface supported by GHC [LP95]. On both machines the program is warm started, i.e. it is run at least once before measurements are taken. Warm starts reduce runtime because the data is preloaded into RAM disk caches in the file system.

Measurements. Figure 6 shows the speedups obtained when the blackspots program is run on both the SPARCserver multiprocessor and the network of workstations. In each graph the top line is linear speedup. The second line is the *relative* speedup, i.e. compared to a single processor running the parallel program. The third line is the *absolute* or wall-clock speedup, i.e. compared to a single processor running the optimised sequential code. The workstation speedups are good, with 16 workstations relative speedup reaches 12 and absolute speedup reaches 10. The 4-processor SPARCserver runtime is significantly less than on the workstations, but the speedups are less impressive, reaching 2.8 relative and 2.2 absolute.

Scaling. In addition to speedups, an important measure for data-intensive applications is *scaleup*, i.e. can a machine twice the size process twice the volume of data in the same time? Figure 7 shows the scaleup for the two machines. There are as many large tiles as there are processors. The scaleup of the workstations is satisfactory: a 44% increase in runtime between 1 and 16 processors. Also note that much of the increase occurs as soon as a second processor is added. Scaleup on the SPARCserver is not nearly so impressive: a 32% increase in runtime with just 4 processors.

4.3.3 Discussion

The GpH blackspots program solves a real problem using real data and exhibits good wall-clock speedups and acceptable scaleup on two very different parallel architectures. The sequential Haskell implementation is an order of magnitude faster than the (interpreted) PFL implementation, and on 16 workstations the GpH program is an order of magnitude faster still.

The simulator and strategies have allowed us to carry out low-cost experiments with several possible parallel variants of the program. The tiled variant is selected for execution on the parallel architectures because it delivers good coarse-grained parallelism under both idealised and realistic simulation. In some ways the parallelism exhibited by this variant is insufficiently irregular to exhibit the strengths of GpH.

The parallelism exploited by the variants of the program is very different. For simplicity we contrast two extremes, by comparing the parallel-pipleline-stages variant with the tiled variant.

The parallel-pipleline-stages variant introduces parallelism within each pipeline stage using a variety of paradigms. The file reading and parsing stage is made data parallel by partitioning the data and reading from n files. Control parallelism is used to construct the accident indices. The stages constructing the same-site relation and the partition both use benign speculative parallelism. A total of 8 strategies are used in the parallel-pipleline-stages variant, some of which are hand crafted. The strategy that speculatively evaluates the first n elements of a list is used twice within the program and may be useful in other programs.

The tiled variant has very simple top-level data parallelism. Essentially the partition function is mapped in parallel over a list of tiles, prior to being aggregated to produce the result. The parallel map function is a standard parallel higher-order. In all the variants parallelisation entails minimal restructuring of the algorithm.

4.4 Naira

4.4.1 Program Description

Naira is a parallel, parallelising compiler for a rich, purely functional programming language. It processes, and its front-end is written in, a subset of the standard Haskell 1.2 language with type classes as the main feature omitted. The front-end comprises about 5,000 lines of Haskell code organised in 18 modules. The back-end is written, following popular tradition, in the C programming language.

The main motivation for writing Naira is to explore the prospects and problems of parallelising a modern functional language compiler [Jun98]. Another aspect is to make the compiler accept parallelised program inputs and to generate multithreaded parallel code so that we can assess the efficiency of the resulting parallel code. These two aspects of Naira — that it is itself parallel and that it generates parallel code — makes it, to our knowledge, the first functional language compiler of its kind. It is also the second largest parallelised Haskell program, following the Lolita natural language parser described in Section 4.5.

The front-end of Naira, which we parallelise, compiles to a graph-reducing parallel abstract machine with a strong dataflow influence. In this section we highlight the structure, parallelisation and performance analysis of the compiler on the GranSim simulator as well as on a network of Sun workstations. A more detailed exposition of the various aspects of the compiler is given in [JDH97] and in the PhD Thesis [Jun98].

The top-level structure of the compiler in terms of the pipeline of its main phases is shown in Figure 8. The first, analysis, pass consists of the lexical analyser and the parser. The next four passes implement the pattern matching compiler, the lambda lifter, the type checker and the intermediate language optimiser, respectively. The detailed organisation and implementation of these passes is described elsewhere [Jun98].

The two-way split after the lambda lifting pass indicates that the result of the lambda lifter can be piped simultaneously to both the type checker and the optimiser and that these latter two phases can proceed in parallel combining their results, using `showModule`, to produce the intermediate code which is input to the code generator.

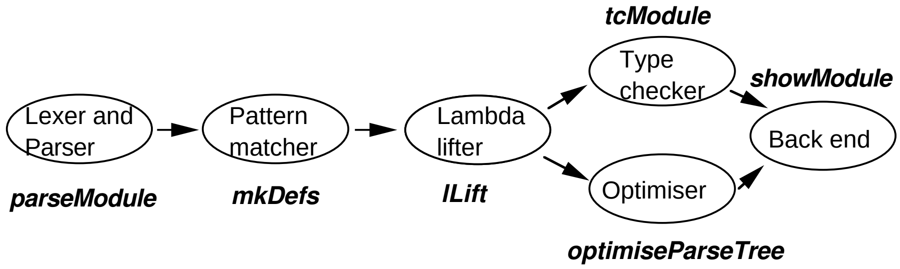

Figure 8: The pipeline structure of Naira's main phases

4.4.2 Parallelising Naira

The compiler is parallelised using evaluation strategies [THLP98] and an allied parallel name-server, which is used to minimise data-dependencies and thus expose more parallelism [JDH97]. The parallelisation proceeded top-down, starting with the top-level pipeline, then proceeding to the lower-levels to parallelise four main passes of the compiler — the pattern matcher, lambda lifter, type checker, and the optimiser — as summarised below.

Top-level Parallelisation. The top-level pipeline is parallelised in a data-oriented fashion by annotating (with evaluation strategies) the intermediate data structures used to communicate analyses results between the the compiler phases. The laziness of the language is crucial here to ensure that the output of one phase is made available incrementally to the next phase(s) so that the analyses in the phases can proceed in parallel.

```
analyseModule fileName modName imports exports symbTabs defs =
    showModule modName impVals dats exports            $||
                                       parPair parForceList parForceList $
    fork (optimiseParseTree fileName exports stOpt  aInfo,
         tcModule fileName stTE exports tInfo  syns) $|| parForceList $
    lLift fileName stPM                              $|| parForceList $
    mkDefs fileName stPM                             $|| parForceList $ funs
  where (stPM,stTE,stOpt)   = symbTabs
        (dats,syns,funs)    = defs
        (aInfo,tInfo,impVals) = imports
fork (f, g) inp = (f inp, g inp)
parForceList = parList rnf
```

Figure 9: `analyseModule` rewritten using Pipeline Strategies

Figure 9 shows the function, `analyseModule`, that implements the top-level pipeline. We use strategic function application, `$||`, to combine the individual passes into a complete program and at the same time define parallelism over the intermediate data structures. The underlined portions show the only code that need to be added to ensure the parallelisation of the top-level pipeline.

Parallelising Individual Passes. The pattern matcher, lambda lifter and the intermediate language optimiser are parallelised, generally, in a data-parallel manner by ensuring that the respective analyses in each phase are applied to all function definitions in a module in parallel. Results of parallelising each of these phases gave only modest speedups of up 2.4 under an idealised

GRANSIM simulation. A more detailed discussion of the parallelisation of these phases is reported in [Jun98].

Cost-centre profiling [SP97] reveals that, as is often the case, the *type checker* is the most expensive part of the compiler, both in terms of space usage and runtime. Therefore, in order to get good overall parallel performance, more attention was paid to the parallelisation of the type inference phase than to the other compiler phases.

The type checker is parallelised using a parallel name server to minimise data dependencies and thus avoid sequentialising the inference process. For instance, to typecheck two quantities d_1 and d_2, we analyse them simultaneously in the current type environment, each returning a type and a substitution record. If a variable v common to both d_1 and d_2 is assigned (possibly different) types t_1 and t_2 from these two independent operations, t_1 and t_2 will be unified in the presence of the resulting substitutions and the unified type associated with v.

Table 4: Performance of Naira with idealised and realistic 8-processor GRANSIM simulations

	Idealised Simulation		**Realistic Simulation**			
			SMP		**DMP**	
	Avg. Par.	**Speedup**	**Avg. Par.**	**Speedup**	**Avg. Par.**	**Speedup**
Best	8.4	8.13	4.9	4.68	5.6	5.32
Worst	1.9	1.40	1.8	1.39	1.8	1.35
Mean	5.5	4.36	4.0	3.95	3.5	3.55

Parallelism has been exploited at four different stages in the type checker:

- in a data-parallel fashion when typechecking definitions in a module;
- in typechecking local definitions in parallel with the top-level ones;
- on calls to frequently used functions; and
- in typechecking aggregate expressions.

The first stage of the parallelisation yields significant parallelism and speedup with the parallelisation of the other stages also leading to modest improvements. Most notably, the overall performance obtained for parallelising the type checker is higher than that obtained after parallelising the top-level pipeline (the latter achieved a mean speedup of 2.4 in an idealised simulation).

Measurements. The compiler has been measured on both idealised and standard setups of GRANSIM simulating both shared-memory (SMP) and distributed memory (DMP) architectures. The results are summarised in Table 4. The idealised simulation achieved a speedup of up to 8.13, with 4.36 as the mean value for all inputs. The results of realistic simulations on a 8 processor machine show a mean speedup of 3.95 in a shared-memory setup and of 3.55 in a distributed-memory setup. The input programs used in the experiments are the compiler's own source modules, 18 in total with 5,000 lines of code. The figures in the table summarise the best, worst and mean results for all modules using idealised, shared-memory and distributed-memory simulations.

Naira has also been measured on a network of Sun workstations (SPARCstations 4/20), running Solaris 2 and connected to a common Ethernet segment. Figure 10 shows the result of measuring Naira on GUM. Overall this figure shows a wall-clock speedup of 2.46, and a relative speedup of 2.73 on a network of five workstations. These results are in agreement with those obtained using GRANSIM which predicted a speedup of 3.01 simulating such a high latency network (this GRANSIM estimate is based on a simulated distributed-memory machine with a latency of 50K cycles).

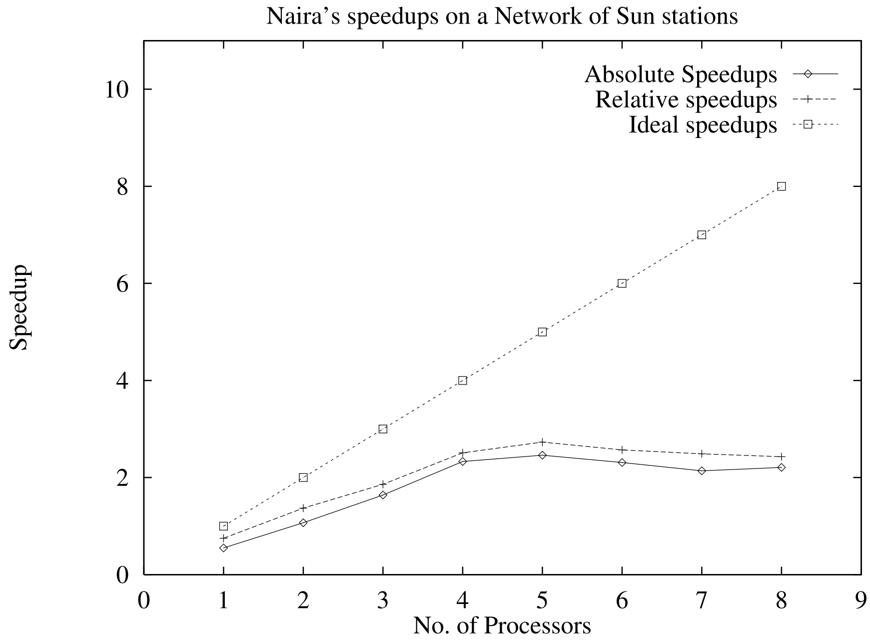

Figure 10: Speedup summary of Naira on GUM

4.4.3 Discussion

At the overall parallelisation stage, where we activated parallelisation code in all the stages, we found that the parallelism measured fell short of the sum of the parallelism figures obtained in the individual stages. This indicates that the evaluation strategies in the different places interfere with one another. Without a more detailed parallel profiler it is quite hard to understand and predict the performance of this rather large program: small changes in the parallelisation code can lead to significant changes in parallel behaviour for some inputs.

Further experimentation with different evaluation strategies could not achieve significant overall performance improvements. This led us to re-examine more closely the algorithms on which the individual phases of the compiler were based. We found that composition of substitutions, which is performed quite often in Naira, forms the main bottleneck in the parallel performance of the compiler. We revised our implementation of this algorithm and fine-tuned our strategic code resulting in substantial performance improvements (see [Jun98] for details).

We have experimented with lists and sorted (unbalanced) binary trees to represent the data structures used in the compiler. Although a tree structure exposes parallelism faster than a list (for the data-parallel processing of the components), the computations needed to maintain the sorting of the trees can be more expensive. Consequently, our experimental results using these representations were, by and large, the same.

Careful study of the parallelism profiles, using the tools of [SP97, HLT97], reveals that file I/O and parsing account for a significant part of the remaining sequential component of the computation and therefore by Amdahl's law represent a major limitation on further optimisation. Parallelising I/O can be quite difficult, and is beyond the scope of the work reported here.

4.5 Lolita

4.5.1 Program Description

This section discusses the Lolita natural language engineering system [MSS94], which has been developed at Durham University. A more detailed presentation of the parallelisation together with measurements of the parallel runtime behaviour can be found in [LMT$^+$97]. The goal of parallelising this application is mainly to reduce runtime but also 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).

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 extract the required information.

Central to Lolita's flexibility is the semantic network, called SemNet, a graph based knowledge representation used in the core of Lolita. In SemNet concepts and relationships are represented by nodes and arcs respectively, with knowledge being elicited by graph traversal. The task of the analysis stages is to transform the possibly ambiguous input into a piece of SemNet. Application-dependent backend stages can then extract pieces of the SemNet and present it in the required form. Currently, SemNet comprises approximately 100,000 nodes or 12Mb.

Since every text has to be translated into a piece of SemNet the parallelisation of this process offers the largest payoff in reduced runtime. Therefore, most of our effort has gone into the parallelisation of this part of the system.

4.5.2 Parallelisation

Pipeline Parallelism. Our immediate goal in parallelising this system is to expose sufficient parallelism to fully utilise a 4-processor shared-memory Sun SPARCserver, our target machine. Following our guidelines for developing parallel programs, we use a pipeline approach to achieve this relatively small degree of parallelism. Each stage listed above is executed by a separate thread. These threads are linked to form a pipeline. In contrast to classical pipelines, which require a large input set to achieve good parallelism, the lazy evaluation mechanism makes it possible to overlap stages of the pipeline operating on the same piece of data.

In order to analyse the parallelism generated by this version it is crucial to understand how this algorithm depends on a lazy evaluation mechanism. The parsing stage generates a forest of possible parse trees. The analysis stages then examine individual trees and pick the most likely tree as the result. Since the analyses in general do not require the full parse tree, it is often possible to avoid generating all of an unlikely tree in the parsing stage, although its probability is determined no earlier than in the analyses stages.

This dynamic behaviour requires special care in the design of the parallel algorithm. It must be guaranteed that no unnecessary parse trees are generated, because sequential profiling indicates that parsing amounts to up to 20% of the overall execution time.

Data-Oriented Parallelism. In order to add data-oriented parallelism to the above program we define strategies on the complex intermediate data structures (e.g. parse trees) which are used to communicate between these stages. This approach simplifies the top-down parallelisation of this very large system, since it is possible to define the parts of the data structure that should be evaluated in parallel without considering the algorithms that produce the data structures. It is not necessary to break the abstraction provided by the sub-functions.

Parallel Stages. Finally, we introduce parallelism in the most time consuming stage, the syntactic parsing stage. Again we have used cost-centre profiling to determine the most expensive stage in the program. The parallelism in this module has the overall structure of a parallel tree traversal. To avoid an excess of parallelism in this stage it is necessary to use a thresholding strategy, which improves the granularity of the parallel threads. This strategy is applied to a system parameter, which reflects the depth in the tree. In fact the same polymorphic thresholding strategy can be applied to two lists of different types.

Speculative Parallelism. Speculative 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.

Combined Parallelism. Figure 11 shows the parallel structure arising when all of the sources of parallelism described above are used. Note that the analyses also produce information that is put into a 'global context' containing information about the semantics of the text. This creates an additional dependence between different instances of the analysis (indicated as vertical arcs). Lazy evaluation ensures that this does not completely sequentialise the analyses, however.

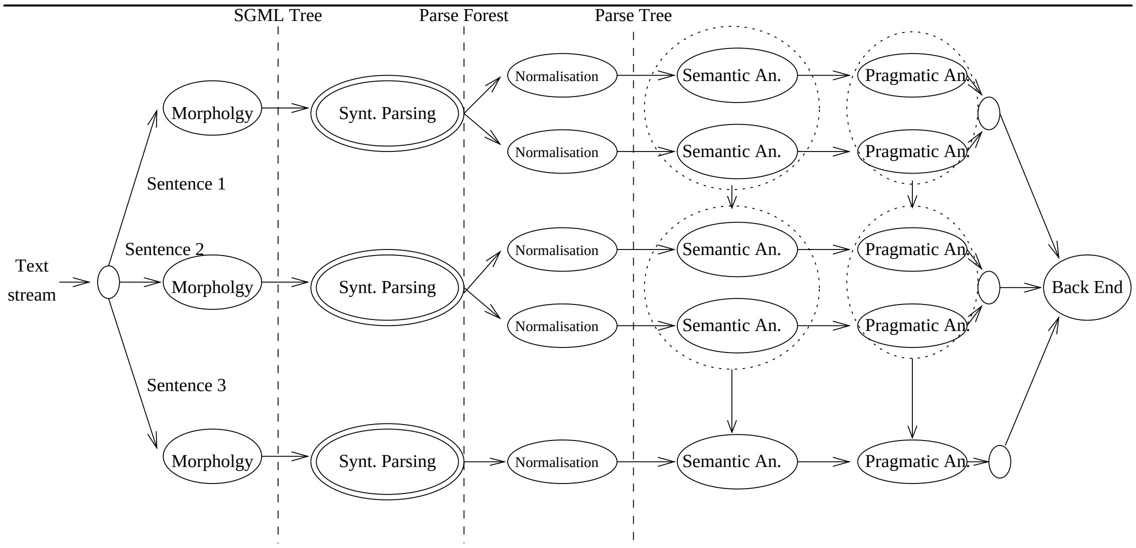

Figure 11: Detailed Structure of Lolita

The code of the top level function `wholeTextAnalysis` in Figure 12 clearly shows how the algorithm is separated from the dynamic behaviour in each stage. The only changes in the algorithm are

1. the use of `parList` in the definition of `rawParseForest` to describe the data parallelism in the parsing stage;
2. the `evalScores` strategy which defines data parallelism in the analysis stages over possible parse trees; and
3. the use of strategic function applications to describe the overall pipeline structure.

The strategies used in `parse2prag` are of special interest. The parse forest `rawParseForest` contains all possible parses of a sentence. The semantic and pragmatic analyses are then applied

```
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 12: The top level function of Lolita

to a predefined number (specified in `global`) of these parses. The strategy that is applied to the list of these results (`parList (parPair ...)`) demands only the score of each analysis (the first element in the triple), and not the complete parse. This score is used in `pickBestAnalysis` to decide which of the parses to choose as the result of the whole text analysis. Since Lolita makes heavy use of laziness it is very important that the strategies are not too strict. Otherwise redundant computations are performed, which yield no further improvements in runtime.

Measurements. Realistic simulations of the pipeline parallel version of Lolita show an average parallelism of 1.6, which is rather satisfactory for only a few top-level changes in the program. The parallelised parsing stage can easily produce several hundred threads. Therefore it is important to tune the thresholding parameter in this stage to avoid excess parallelism. We have not systematically measured the possible improvements in the quality of the result that should be possible by the speculative parallelism described above. A more detailed discussion of the parallel variants of Lolita is given in [LMT+97].

A realistic simulation of Lolita showed an average parallelism between 2.5 and 3.1, using just the pipeline parallelism and parallel parsing. Since Lolita was originally written without any consideration for parallel execution and contains a sequential front end (written in C) of about 10–15%, we are pleased with this amount of parallelism. In particular the gain for a set of rather small changes is quite remarkable.

In contrast, under GUM with two processors and small inputs we only obtain an average parallelism of 1.4 (see Figure 13). With more processors the available physical memory is insufficient and heavy swapping causes a drastic degradation in performance, which prohibits any wall-clock speedup. 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 (totaling 16Mb), and requires 100Mb of virtual memory in total in its sequential incarnation.

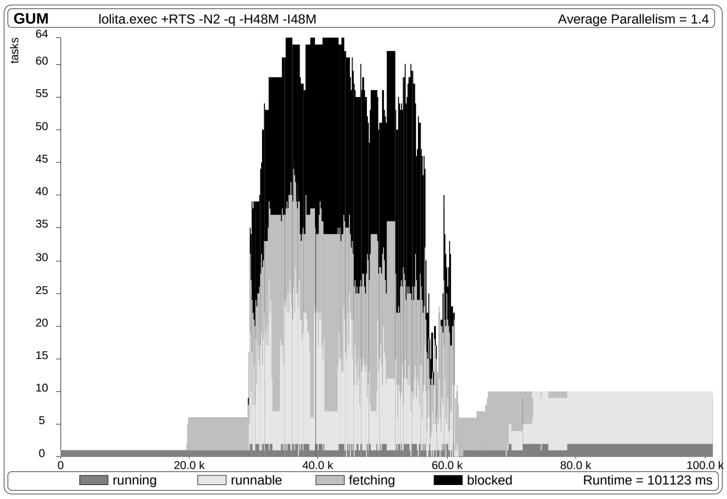

Figure 13: Activity profile of Lolita run under GUM with 2 processors

Figure 13 shows the activity profile of running Lolita under GUM with 2 processors. The sequential front end in Figure 13 is caused by the sequential part of the parsing process. The middle third of the graph shows a high degree of parallelism generated by the parallelised parsing stage. In this setup we have tuned the thresholding parameter to produce only a small amount of parallelism to avoid high memory consumption, which is the main reason for not achieving further reductions in runtime when using a 3 or 4 processor setup. In the final third of the execution the pipeline parallelism of the analysis stages generates a good utilisation of the machine.

4.5.3 Discussion

The most intriguing aspect in the parallelisation of Lolita is that the parallelism is achieved using a very small number of changes to the Haskell parts of the application. We have been able to use a top-down approach of the parallelisation to an extent, which would be very difficult in a strict language. All of the parallelism has been specified by evaluation strategies acting on the data structures passed between modules. As a result, the parallelism has been introduced without changing, and indeed without understanding most of the program. This abstraction is crucial when working on an application of this size. For example, introducing top-level parallelism entailed changing just one out of around three hundred modules.

We have used speculative parallelism in order to improve the quality of the results. This underlines the importance of speculative parallelism, which we have already seen in parallelising the Alpha-Beta algorithm. The integration of the C code into the parallel version complicated the parallel algorithm because foreign language calls implicitly fully evaluate their results, bypassing the strategic description of the dynamic behaviour. Finally, we have found a need for limited support of persistence. The SemNet is a conceptually persistent data structure, because it is required by every invocation of Lolita. In the absence of support for persistence the current code uses foreign language calls to achieve efficient I/O. Again, these calls interfere with the strategies defined in the program.

The achieved average parallelism of Lolita lies between 2.5 and 3.1 under GranSim emulating a 4-processor shared-memory machine. The corresponding speedup, however, does not exceed 2.4. This is partly due to overhead caused by very fine-grained parallelism and partly due to strategies that perform speculative computations (although we avoided speculation on potentially expensive components). The GUM version does not achieve significant wall-clock speedups, yet. This, however, is not due to a lack of parallelism but due to the very high memory consumption of the application, which exceeds the available main memory in the current setting.

4.6 LinSolv

4.6.1 Program Description

The linear system solver that is discussed in this section, and in more detail in [Loi97], is an application from the area of symbolic computation and uses an approach that is very common for computer algebra algorithms: a multiple homomorphic images approach [Lau82]. The main idea of this approach is to solve a problem in a set of simpler domains, called homomorphic images, and then to reconstruct the overall solution from the solutions in the individual domains.

In the case of the LinSolv algorithm the original domain is $\mathbb{Z}$, the set of all integer values, and the homomorphic images are the domains $\mathbb{Z}_p$, the set of integers modulo 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 [Lip71]. This overall structure of the algorithm is shown in Figure 14.

In the solution phase we use an algorithm based on Cramer's rule, which describes how the components of the result vector can be computed as the ratio of two determinants. Although this algorithm is less efficient than alternatives like Gaussian elimination 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 determinant computation itself can be parallelised using a divide-and-conquer structure.

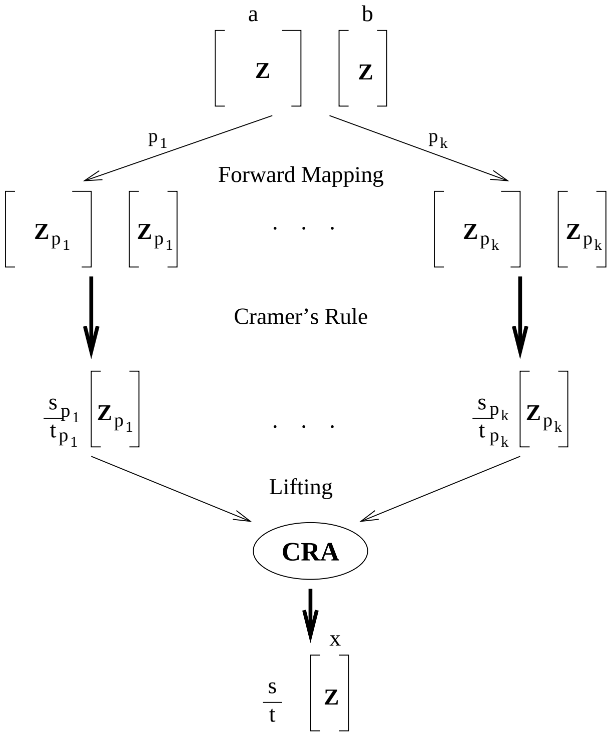

Figure 14: Structure of the LinSolv algorithm

Figure 15 shows the top level of the LinSolv algorithm. 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 determinant in the homomorphic image generated by the prime p is 0. In this case the result cannot be used in the lifting stage in order to compute the overall solution. The corresponding prime number is termed unlucky.

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` is used. The following section discusses a strategy that describes a parallel version of this algorithm.

4.6.2 Parallelisation

Algorithm. In the parallelisation of this algorithm it is important to define evaluation degree and parallelism over the infinite list `xList`. Without controlling the parallelism on this data structure the CRA will demand each solution sequentially, because the most efficient version of the CRA uses a list fold operation.

The definition of `strat` in Figure 16 represents the final strategy in the performance tuning of the algorithm. In order to avoid a dependency between the solution phases, this strategy guesses the number of primes needed to compute the overall result and uses a `parListN` strategy to generate data parallelism over an initial segment of the infinite list `xList` of the solutions in all homomorphic images. Using `parList` inside the `par_sol_strat` strategy causes each component of the result to be evaluated in parallel. However, it is necessary to check whether the homomorphic image of the original matrix is zero to avoid redundant computation if the prime is unlucky. In order to minimise data dependencies in the algorithm we do not already check the determinant

```
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 15: Top level code of the LinSolv algorithm

```
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 16: Strategy of the parallel LinSolv algorithm

when computing `noOfPrimes`. If some primes 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 xs` 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.

Measurements. In developing this parallel algorithm we have used GranSim in a realistic setup, simulating a closely-connected 32 processor machine. Whereas earlier versions showed bottlenecks at some points during the computation, the activity profile for this final version in Figure 17 shows a consistently high degree of parallelism.

Our measurements of LinSolv under GUM on a 3 processor shared-memory machine correspond to the behaviour predicted by the GranSim simulator. We achieved relative speedups of up to 2.1 and absolute speedups of up to 1.7. More details of these measurements can be found in [Loi97].

In the performance tuning of this algorithm the visualisation tools have been crucially important. Early parallel versions of the algorithm showed bottlenecks caused by the sequential demand on the solutions generated by the list-structured lifting phase. This behaviour resulted in a sequence of parallel executions with regular drops in-between. The code in Figure 16 avoids

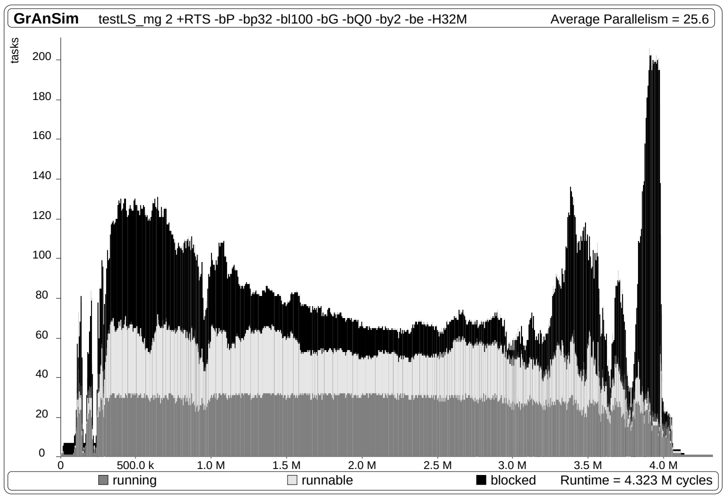

Figure 17: Activity profile of final LinSolv

this bottleneck by guessing the number of primes that are needed and by using data parallelism via a `parListN` strategy. A more detailed discussion of the performance tuning of the parallel algorithm is given in [Loi97].

4.6.3 Discussion

Several properties of evaluation strategies have been important in parallelising the algorithm. We made use of strategies being higher-order to describe nested parallelism: an outer strategy defines the parallelism over `xList` with a strategy `par_sol_strat` as argument that defines the parallelism over the elements of this list. Thereby the strategy reflects the nested data-structure over which the parallelism is defined. The separation between algorithmic and behavioural code made it possible to experiment with different versions of the parallel code, without changing the algorithm. This was very important during the performance tuning of the algorithm. It is worth noting that all parallelism can be described on top level, unlike in the pre-strategy code where a lot of the parallelism was defined in sub-functions.

The strategy in Figure 16 also demonstrates how conservative parallelism can be defined over an infinite data structure. There is no need to rewrite the algorithmic code that generates the data structure in order to express a degree of parallelism that does not generate any speculative computation.

The development and performance tuning of LinSolv predated the design of evaluation strategies. This gives us the possibility to directly compare the pre-strategy with a strategic version of the code. The pre-strategy version of the code combined the computation of the result with a specific dynamic behaviour suitable for parallelism. For example a tree-structured CRA algorithm has been used in order to force the computation of the individual solutions independently. Because some homomorphic images may turn out to be not suitable for computing the overall result, a separate "fail handler" had to be used in order to compute more results if necessary. The resulting control parallelism yielded rather opaque code with parallelism defined in one sub-function, namely the CRA. In contrast, the strategy version only uses data parallelism and cleanly separates the parallelism from the algorithmic code.

The multiple homomorphic images approach is used in many computer algebra algorithms such as resultant computation [HL94] and p-adic computation [LL93]. It should be possible to use the same overall structure of parallelism for these versions, only replacing the function that guesses the number of primes and the strategy defining the inner parallelism. In this case the polymorphism

Table 5: Results summary

Program	Lines of code	Wall-clock speedup on few procs (arch:no. procs)		Simulated speedup (no. procs)		Best wall-clock speedup (arch:no. procs)	
Blackspots	1,300	3.14	(WkStn:4)	3.7	(4)	10.00	(WkStn:16)
Blackspots	1,300	2.16	(SMP:4)	3.7	(4)	2.16	(SMP:4)
Naira	5,000	2.33	(WkStn:4)	3.0	(4)	2.46	(WkStn:5)
Lolita	47,000	0.90	(SMP:2)	2.4	(4)	0.90	(SMP:2)
LinSolv	800	1.66	(SMP:3)	2.3	(4)	1.66	(SMP:3)

of strategies enables a code reuse for defining parallelism.

5 Program Comparison

Where the previous section described the implementation and measurement of individual programs, this section discusses common aspects of the programs. We focus on the parallel paradigms used in the programs, and the large-scale issues encountered. We also summarise the results allowing approximate comparison.

5.1 Comparative Measurements

The most significant result of this paper is that we are able to achieve modest wall-clock speedups for all of the programs, except Lolita. The simulated speedup for Lolita is good, and we believe that it is only limitations on physical memory that prevent a wall-clock speedup of Lolita.

It is also important to emphasise that the programs have been measured on several parallel systems, utilising different ports of the GUM runtime-system. In a separate paper [TBD+98] we focus on this aspect of architecture-independent parallelism, and its practical impact on the development of parallel GPH programs. The following measurements are based on networks of workstations and shared-memory multiprocessors, as detailed in Section 4. The systems represent two very different classes of parallel architectures: shared- and distributed memory machines. The wall-clock speedups on both architectures underline the flexibility of our parallel programming system.

Table 5 summarises the results for each program, and the columns are interpreted as follows. The first column gives the program name. The second column gives the approximate number of lines of source-code, including libraries. The third column is the wall-clock speedup of the program on a small number of processors, together with the number of processors and the parallel architecture — a network of workstations (WkStn) or a shared-memory multiprocessor (SMP). Wall-clock speedup is measured by dividing the elapsed time for parallel execution by the elapsed time for the same program compiled and optimised for sequential execution. The fourth column gives the simulated speedup achieved under GRANSIM emulating the target architecture. The last column gives the best wall-clock speedup achieved, together with the number of processors used and the architecture.

The Blackspots program achieves the greatest wall-clock speedup, but although it uses some complex algorithms, it has a simple data parallel structure, and only a small amount of irregularity in the thread sizes. Although the speedups for the Naira compiler are smaller, it more truly represents the class of programs that we expect GPH to be used for. That is Naira as a complex symbolic computation with an elaborate parallel structure. Lolita is similar in being symbolic and having an irregular parallel structure. It is also very large and multi-lingual (Haskell and C). Unfortunately, while a realistic simulation of Lolita delivers good speedups, exhibiting a large amount of inherent parallelism, the wall-clock figures are poor because of the high resource utilisation. LinSolv is symbolic, and has irregular parallelism defined over a potentially-infinite data structure. It delivers modest wall-clock speedups on a shared-memory machine.

5.2 Parallel Paradigms

The programs use a number of parallel paradigms, often nesting one paradigm inside another. For example both Naira and Lolita nest a pipeline within a data-parallel paradigm. Version II of the Blackspots program is still more elaborate having a pipeline with stages using data-parallelism, control-parallelism, and benign speculation. The following parallel paradigms have been used in the development of the parallel algorithms discussed in this paper:

- **Data parallelism**: Naira, Lolita.
  In the data parallel paradigm every element of a data-structure is evaluated in parallel. Naira is data parallel over the function definitions in a module. Lolita is data parallel over the sentences in the text.

- **Pipeline parallelism**: Naira, Lolita.
  In the pipeline parallel paradigm a sequence of stream-processing functions is composed together, each consuming the stream of values constructed by the previous stage and producing new values for the next stage. Pipelines in a non-strict language are very flexible over the data type they operate on and have fine-grained parallelism. That is, a pipeline can be defined over any data-structure passed between stages, e.g. both Naira and Lolita pass forests of trees between pipeline stages. The fine granularity means that the producer and consumer may synchronise on every node of a data structure, or the producer may construct all of the structure before any of it is consumed or, more likely, something in-between. As a result of this fine granularity, pipelines in a non-strict language can be effective even for small input data sets.

- **Task Farm**: Blackspots (Version III).
  In the task farm paradigm a 'farmer' process has a collection of tasks, and 'worker' processes obtain a task from the farmer, and on completing it, obtain another. In Blackspots the task farm has a special form because each task is to evaluate some data structure, and such a farm is more accurately termed a data farm [MS95].

- **Divide-and-conquer**: LinSolv, Lolita.
  In the divide-and-conquer paradigm the problem to be solved is decomposed into smaller problems that are solved in parallel and the solutions are recombined to produce the result. It is easy to generate a great deal of parallelism with this paradigm: the number of tasks is exponential in the number of division steps. The unfortunate corollary is that there may be a large number of very fine-grained tasks generated. We maintain a good thread granularity by including a threshold in the strategy that ensures that small tasks are not sub-divided but evaluated sequentially.

- **Speculation**: Alpha-Beta, Blackspots (Version II), Lolita.
  GpH does not support general speculation, e.g. speculative and mandatory threads are not distinguished, and there is no mechanism for killing unwanted speculative threads. We do, however, use a restricted form of speculation, which we term *benign*. The restriction is that the speculative threads must perform only a small amount of work and create no new threads. Often speculation is controlled by a parameter of the speculative strategy, and selecting an appropriate value is crucial to avoid wasting resources [LMT+97]. It is interesting that several of the programs use speculation because it is a technique that cannot easily be introduced by automatic parallelisation methods.

Some parallel paradigms not explored in these programs include branch and bound, SPMD, bounded buffer and general speculation. We have strategies, and some toy examples, for bounded buffers and SPMD. It appears that general speculation and branch and bound are more problematic within GPH.

Another important aspect of the parallel runtime-system is dynamic load management. It has previously proven to be essential for obtaining good speedups on some programs executed on the GRIP architecture [HP92]. In the context of GUM the importance of dynamic load management is best reflected by the final version of the Blackspots program. This version uses dynamic load management to obtain an even load when evaluating the tiles of a geographically partitioned data set.

5.3 Large-Scale Issues

In the implementation of the programs we encounter a number of aspects of parallel programming in-the-large.

- Application-specific strategies can be rather easily reused in large applications. One example is the merging of lists of a polymorphic type in Lolita, which is used in two places. Clearly, the polymorphic nature of the language aids code reuse in this case.

- Some of the programs were made parallel by someone other than the original author, most notably Lolita. In these circumstances the largely-implicit parallel programming model is crucially important, because parallelisation does not require the explicit introduction, and synchronisation, of threads. Instead parallelisation is similar to sequential performance tuning in that it entails understanding time and space consumption, data dependencies, and often controlling evaluation degree. In that sense parallelisation does not add a new dimension of complexity to the program design, it merely complicates the existing process of performance tuning. We believe that it would be much harder to parallelise a second author's program using an explicitly parallel programming model.

- Parallelism can be described at a high-level, and this means that only a small part of a large system needs to be understood, changed, and recompiled. For example adding parallelism entails changing just two out of three hundred modules in Lolita, and one out of five in Blackspots.

- The parallel version of a large program may have very large resource utilisation. This is likely to be a problem on shared-resource machines, e.g. multi-processors with shared memory or disks. For example the sequential variant of Lolita uses 100Mb of heap, and the parallel variant needs approximately 64Mb per processor. Similarly, in Blackspots every processor initially reads a file, generating intense network and disk traffic.

- A major task in parallelising a large program is to define basic strategies over the data types, in particular a strategy to reduce values of the type to normal form (`rnf`). Fortunately the `rnf` function can be derived automatically from the type, and we have constructed a tool that allows us, *inter alia* to automatically add basic strategic definitions to a module [Win97].

- Strategies may also be required over library data types, e.g. `parSet`. Unfortunately this entails using a private copy of the library module.

- A GPH program can be used to prototype alternative parallelisations of an imperative program. Experimenting with alternative parallelisations is easier in GPH than in imperative languages. Parallel prototyping has been used in LinSolv to tune the algorithm.

- Many of the programs had been written without the intention of making them parallel, e.g. Naira and Lolita. It is still possible to obtain parallelism, albeit modest, without restructuring these programs.

6 Evaluation of GPH Programming

In this section we reflect on our experiences programming in GPH, i.e. in a functional language with largely-implicit parallelism. We both analyse and consider future directions for the language, the co-ordination mechanism (evaluation strategies) and the programming environment.

The most important language result is that despite the apparent tension between parallel an lazy computation, they can be usefully combined to produce a programming model with a high degree of modularity. This modularity is due to the *data-oriented* style of programming offered by a lazy parallel programming model. This means that it is sufficient to define the parallelism only on a few crucial data structures, which typically are passed between sub-functions at the top level of the program. Because lazy evaluation delays the generation of the result until it is needed, strategies can be used to define evaluation degree and parallelism outside the function generating the data structure. This achieves a level of modularity not encountered in languages with a strict evaluation mechanism. Most importantly, the programmer can define the parallelism without breaking the abstraction of individual functions, which is an important property for large programs where the parallelisation is probably not performed by the author of the program.

In optimising the performance of programs written in a language with lazy evaluation it is sometimes difficult to track the implicit evaluation degree of some data structures. This stems from the high-level semantics of the language, which makes it more difficult to track the operational behaviour of the program execution. Although this property allows the programmer to write shorter and more flexible programs, it poses a difficulty when optimising the code. With our model of parallel lazy evaluation, evaluation strategies can be used to make evaluation degree explicit where required, and thus to enforce a certain operational behaviour.

In short, evaluation strategies can be used for both sequential and parallel performance tuning. In this sense, parallelisation is just a refinement of the performance tuning process, which offers even faster computation. Most notably, however, there is no need to extend the underlying programming language by e.g. introducing an explicit notion of threads. Our experiences with the use of evaluation strategies on large lazy functional programs indicate that a lazy parallel programming model offers the prospect of cheap, modular parallelism with only a minimal coding effort.

6.1 Language

The parallel language we are using, GPH, is only explicit in exposing parallelism in the source code. The management of the parallelism is completely hidden by the runtime-system. In this approach many classical problems of concurrent programming such as generating deadlocks or race conditions between threads do not arise. However, it is still possible to tune the parallelism by specifying the size of the parallel computation and the evaluation order.

The features of the language that we found to be most important are as follows.

- **Determinism** makes parallel program development easier because the algorithmic part of the program can be developed in a sequential context. Inserting strategies to introduce parallelism does not change the value computed, and will not change the termination conditions as long as the strategies are no more strict than the original function, i.e. the parallelism is conservative.

- **Largely-implicit parallelism** ensures that only a small amount of additional code is required to introduce parallelism. In particular, it is only necessary to expose parallelism, by marking expressions.

6.2 Evaluation Strategies

For any program, the primary benefits of the evaluation strategy approach are similar to those that are obtained by using laziness to separate the different parts of a sequential algorithm [Hug89]: the separation of concerns makes both the algorithm and the dynamic behaviour easier to comprehend and modify [THLP98].

In large programs, strategies allow us to raise the level of abstraction because the programmer introducing parallelism need not understand the low-level details of the whole program. Strategies allow us to

- **Describe top-level parallelism**. Often some initial parallelism can be obtained by parallelising the top-level of the program with a very shallow understanding of the algorithms used in the program.
- **Preserve module abstraction**. Parallelism can often be specified on the data structures passed between modules. The programmer need only know which items of the data structure can be computed independently, which is often simpler than understanding the algorithm used to compute them. Indeed the type of the data structure may even give a hint on which strategy to use for parallelising the program.

This style of programming offers a level of abstraction to the programmer that does not exist in parallel imperative languages. However, if it is necessary, the evaluation can be controlled in more detail, yielding parallelism described on a similar level as in more conventional parallel programming models.

The presented programs use the power of strategies. In most of the programs strategies are defined over many types, program-specific strategies are constructed, and some of the new strategies are created by composing existing strategies. The specific features that proved most useful are mainly the high-level constructs. Many of the strategies are:

- **Polymorphic**. Strategies that can be used at many types are easier to re-use, for example the polymorphic `mergeStrategy` strategy is re-used in Lolita.
- **Parametric**. The behaviour of a strategy can be modified by parameters. For example the number of elements of a list to evaluate in parallel is a parameter in the Blackspots program, and the similar `force-length` parameter in Alpha-Beta.
- **Higher-order**. This is particularly useful when a strategy takes another strategy as a parameter, thus capturing a class of behaviours as determined by the argument strategy. In LinSolv, for example, a list strategy is passed to another list strategy to describe parallelism over a list of lists. Nesting strategies in this way is a natural means of achieving nested parallelism.

Finally it should be noted that evaluation strategies must be used with care to avoid conflict and malignant speculative computations. The latter can yield higher parallelism because of the extra speculative computations but can also adversely affect a program's completion time. For example generating more possible syntactic parses in Lolita would produce more speculative parallelism, because each of the parse can be analysed in parallel, but it would not reduce the total runtime, because only the best result will be chosen at the end.

6.3 Programming Environment

It has proved essential to develop the programs in a rich execution environment. Several programs were initially developed using the Hugs interpreter, where the interactive mode facilitates debugging. All programs were run under GHC's sequential runtime system. Almost all of the programs used time and heap profiling to identify computationally-intensive components.

To develop the parallelism the programs are first run under GRANSIM to produce idealised, and then realistic simulations. We find that visualising the parallel execution in several ways is essential to the programmer's understanding, and hence improving, the parallelism. The most useful means of visualising the execution are activity profiles like Figure 17 and thread granularity profiles, which show the total runtimes of the individual threads as a histogram.

Using GUM the parallel performance of the programs is measured on a number of platforms. Some of the programs are measured on a network of workstations, e.g. Naira. Other programs are measured on a shared-memory SUNserver, e.g. Lolita. Blackspots has been measured on both workstations and SUNserver. It is unusual to have both shared- and distributed-memory measurements for a single program. We discuss the architecture-independent nature of GpH programming in detail in [TBD+98].

7 Related Work

In his 1993 thesis [Cla93] concerning the implementation of a large parallel rule-based interpreter written in Haskell, Clayman observed with some chagrin that

> "the current facilities for executing functional programs in parallel environments are not effective for large applications. The use of hand-coded annotations may be fine for small programs but it is unsuitable for large programs. Furthermore, there is a lack of parallel systems on which programs can be executed."

Clearly, in the last 5 years some considerable progress has been made towards addressing the criticisms raised in Clayman's thesis. In our own setting we have:

- demonstrated that it is possible to write large parallel applications in Haskell;
- introduced *evaluation strategies* [THLP98] to allow simple and flexible control of parallel programs, so addressing Clayman's criticism of hand-coded annotations; and
- produced an implementation based on standard portable message passing libraries, so vastly extending the number of parallel systems on which our programs may be run.

Although our work is not isolated, and other groups have produced systems that possess similar characteristics to those we espouse (e.g. Sisal [Ske91], NESL [Ble96], Concurrent Clean [NSvP91], Id [Nik91], or Paralation Lisp [DGF97]), Clayman's criticisms do still apply to some extent in a general setting, however. Despite the fact that many parallel implementations of functional languages have been produced, there are relatively few systems that have been developed beyond the prototype stage, and fewer that can also claim to demonstrate architecture independence. Those that can make this claim have been surveyed in an independent paper [TBD+98].

This section surveys existing large parallel functional programs which, like those introduced in this paper, either form complete real end-user applications or are realistic in being taken from a real application domain rather than artificially designed to demonstrate some benchmarking issue. We have therefore excluded such benchmarks, unless they form part of some larger, more interesting application.

The term "large" is not precisely defined, of course – we have taken it to mean over about 500 lines of functional code (which corresponds to an imperative program of some 1,500-5,000 lines). For comparison, all the applications described in this paper apart from the alpha-beta search algorithm comprise more than 800 lines of code each. Unlike the Lolita program which was described earlier, however, the majority of the applications presented here are not large in a strict software engineering sense, since they have been written by single users rather than as large collaborative projects.

The applications described in this section cover a wide variety of problem domains, from numerical applications written in Sisal [Ske91, Can92] or NESL [Ble93] to theorem provers [RW95] and real-time commercial telephony systems [Arm96]. We have not, however, attempted to cover individual implementations or language constructs in depth. The interested reader is referred to the more general literature on parallel functional programming for coverage of these and other significant issues (e.g. [Ham94, HM98]). The most closely related approaches to parallelisation, our earlier work on the FLARE applications [RW95] and the Dutch Parallel Reduction Machine project [BvH+87], are briefly surveyed in Section 7.7.

7.1 Compilers and Rule-Based Systems

While Naira is unique, as far as we know, in being the first complete functional language compiler to have been parallelised [Jun98], there have been a few parallel systems with similar characteristics.

Clayman's thesis described one such application: a functional version of the OPS5 rule-based system that is often used to implement expert systems [Cla93]. This application has a similar structure to Naira, comprising a rule compiler plus production matcher and evaluator. The rule compiler includes pattern-matching and other components. The production matcher and evaluator are best regarded as being analogous to Naira's runtime-system.

Unfortunately, as hinted above, despite mapping out the parallelisation process that he intended to pursue, Clayman was ultimately frustrated by the state of the compiler and implementation technology in 1993, and therefore never achieved his goal of successfully parallelising his program. We are therefore deprived of a potentially interesting comparison between two similar applications. We hope that we are now in a position where Clayman's work could be completed in order to allow a good comparison between these systems.

While not directly usable as part of the compilation process itself, Boucher and Feeley have constructed a parallel implementation of an LR(0) parser generator in MultiLisp [BF94]. The parallelisation process involves the creation of all reachable states in parallel. Simple locks are used in place of the sequential hash table to prevent several tasks working on the same state simultaneously, and to ensure atomic update for each state.

Overall, the parser generator achieves an absolute speedup of 10.4 on 32 processors. The parallel overhead was particularly serious for this system, generating a slowdown of a factor of 3 on one parallel processor, so this represents an impressive superlinear relative speedup (a factor of 33.6 on 32 processors). Given that the overhead exists in the one-processor case, and that the algorithm exhibits super-linear speedup, it seems unlikely that this overhead is simply a consequence of poor locality, as the authors suggest. The super-linearity is claimed to reflect decreased garbage collection costs in the parallel implementation.

Finally, although it has not yet been executed on a parallel machine as far as we are aware, the Id in Id compiler from MIT is, of course, parallel in principle. Id is untyped so the parallel type inference algorithm that gave effective performance improvements in the Naira compiler would be of no direct use (it might conceivably be exploited for e.g. code generation, however). Work we have done in relieving dependencies in the Naira symbol table and pipeline stages seems likely to find a counterpart in any parallel version of the Id compiler, however.

Theorem Provers

There have been several attempts to parallelise functional theorem-provers. As part of the FLARE project [RW95], Hanna and Howell parallelised the 8500 line tautology checker that forms the core of the Veritas theorem prover. This parallelisation was achieved using only the basic `par` and `seq` combinators described earlier. Granularity control was introduced using thresholding based on the size of the propositions to be checked. Performance results for the GRIP multi-processor showed that an absolute speedup of a factor of 18 could be achieved on 20 processors. Work on this application and others from the FLARE project motivated the design of evaluation strategies to help simplify the parallelisation process.

There have also been several implementations of the Boyer-Moore theorem prover. For example, Sodan and Bock's automatically parallelising Lisp system, ParLisp, has achieved a simulated speedup of between 5.1 and 29.5 on an idealised configuration of the MANNA machine containing an infinite number of processors [SB95]. In conducting these experiments Sodan and Bock observe that it is important to check the potential parallelism of the application before proceeding along an expensive implementation route. This is in accordance with the methodology we have propounded both in this paper and elsewhere [THLP98], of using first an ideal simulation to demonstrate parallel feasibility and then refining the simulation to deliver more accurate information for particular classes of target architecture.

The Boyer-Moore theorem prover has also be implemented in Id as part of the Impala benchmark suite [Sha98], but we are not aware of any parallel performance results that can be used for comparison.

7.2 Image Processing

Graphical applications are obvious candidates for parallelisation. While imperative parallel graphics applications generally depend on partitioning (updatable) arrays, more sophisticated data structures may simplify the partitioning process and offer better long-term opportunities for parallelism. Several applications have been produced that perform complex graphical manipulations, including ray tracing to determine the intensity of light that falls on an object, and the computer vision applications prototyped by Michaelson and Scaife in Standard ML.

Ray Tracing

The simple ray tracer that was originally developed in Kelly's thesis for the Caliban coordination language [Kel89] has formed the basis for a number of subsequent studies, including as one of the FLARE applications described above. In the latter case we were able to demonstrate good speedup for this application running on GRIP under a variety of conditions, achieving an absolute speedup of 10.5 on 17 processors, with no evidence of a software performance bound [HMP94]. Relative speedup for the same configuration was a factor of 14.

In his thesis [Tay97], Taylor studies this same ray tracer in the context of Advanced Caliban. Advanced Caliban extends the Caliban coordination language in a number of new and interesting ways that parallel the development of evaluation strategies (for example, the use of nested `moreover` clauses to control placement is similar to our use of strategies to describe process structures). Unlike evaluation strategies, however, Caliban remains firmly routed in a static model of process placement, and the target architecture is restricted to a distributed, closely-coupled parallel machine (in Taylor's case, the 48-node AP1000 at the Imperial College Parallel Centre, London). Using a static process farm, with limited speculative evaluation, Taylor achieves a relative speedup of 17 on 35 processors for this implementation of the ray tracer. With the introduction of manual granularity control, performance can be boosted to a relative speedup of 24 on 35 processors. This is broadly in line with the GRIP results cited above, though speedup is slightly lower.

Bratvold also studied the performance of the ray tracer application [Bra94] using his automatically parallelising skeleton-based compiler for SML, SkelML. Bratvold's thesis results show that a speedup of 9.5 on 22 Transputers could be achieved for the largest example that was tried. In contrast to the dynamic approach we have used in our implementation and in accordance with the Caliban philosophy adopted by Taylor, Bratvold's approach uses a static cost-modeling step to guide the choice of skeleton from a fixed library.

Kesseler also used the ray tracer as a benchmark for Concurrent Clean [Kes95, Kes96]. Kesseler's system adopts a similar skeleton approach to that taken by Bratvold, and also targets a Transputer system. Kesseler reports a speedup of 10.0 on 16 processors, rising to 33.5 on 64 processors, where he is clearly encountering some performance bound. From our own experience, we conjecture that this may be due to poor distribution as a consequence of static process allocation.

While it is hazardous to compare only speedup and not look at absolute performance, it is interesting that the systems using static placement do not exhibit better speedup results than the system of dynamic placement used in GRIP. This is, of course, partly due to the lower communication latencies that apply in GRIP hardware. However, we feel it is a strong indicator that our model of dynamic process placement can yield good parallel performance whilst requiring rather less programmer effort than precise static placement, despite the greater overheads of dynamic control.

Parallel Vision

Michaelson and Scaife [MS95] describe the implementation of several components of a parallel vision system. The overall purpose of the system is to recognise 3D objects in a 2D scene by using information about the relative intensity of light throughout the scene. The parallel algorithms are prototyped using a skeleton-based SML implementation, before being translated to Occam and executed on a distributed-memory Meiko machine (based on Transputers). The SML prototype required 1700 lines against the 3000 lines of the final OCCAM implementation. It was used to verify the general line of parallelism to be taken in the final implementation, in a similar way to our own simulator-based proofing steps.

The primary algorithm used in this application is the Hough transform for solving sets of underdetermined equations. This is parallelised in a data-oriented fashion using a farm skeleton to realise a parallel map over a nested list. Performance was optimised by splitting the data into more sets of equations, so introducing more small tasks which can be managed more efficiently to improve the overall load balance. This confirms our own observations concerning task granularity [LH95] as well as theoretical analyses [BR94]: finer-grained programs are much easier to manage dynamically, and result in much better balanced computation. Overall, Michaelson and Scaife achieve an absolute speedup of 10.5 on the 30-processor Meiko. This performance was less than hoped for, possibly as a consequence of poor load-balancing and/or high communication costs that may arise from the nature of the farm skeleton, which will tend to introduce communication bottlenecks to the farming processor.

Mitrovic and Trobina have implemented some components of a computer vision system in Sisal [MT93]: specifically the Gaussian smoothing and Canny edge detector algorithms that are also used by Michaelson and Scaife. The Sisal program was about 300 lines, compared with 600 for the C version, and took 2 days to write, compared with about a week for the C program. The final stage of the vision system (image compilation) was however slightly larger than the corresponding C program (600 lines versus 500). Overall the Sisal program ran 10% faster than the C program when run sequentially and achieved a relative speedup of 3.1 on a 4-processor shared-memory SGI machine, without requiring further coding effort. This is clearly a very creditable performance gain for such programmer modest effort. Similar performance results have been verified by other Sisal applications [Can92], some of which are described below (Section 7.4).

7.3 Data Intensive Applications

There have been relatively few attempts to produce large-scale data-intensive functional applications, and even fewer that have been successfully parallelised. One of the most interesting is the AGNA system, which implements read-only selections (lookups) over a parallel functional database [HN91].

AGNA

The AGNA system uses list comprehension to structure read-only queries over an on-disk database. Since each lookup is independent of the results of any other lookup, parallelisation is straightforward and very high parallelism can be achieved with a good prospect of scalability. Heytens and Nikhil [HN91] report a speedup of 31 on a 32 processor distributed-memory machine for non-indexed lookup. Indexed lookup is much faster, but speedup is limited to a factor of 8, due to task creation and result construction costs in the implementation that was adopted.

Parade

As part of the EPSRC Parade project we have investigated parallel functional database transaction processing where the transactions involve not simply queries, as with AGNA, but also update operations that may introduce dependencies with subsequent database transactions [AHPT93]. Our results show that acceptable parallel performance can be achieved through the use of techniques to reduce the "hot-spot" that arises from contention on the root of the B-tree data structure that forms the index to the on-disk database. Overall, we achieved an absolute speedup of 12.6 on 15 GRIP processors. Larger data sets gave better performance than smaller ones, so it seems likely that these results could be scaled to larger systems with higher throughput. Unlike AGNA, our results apply only to in-memory copies of the database, however, with simulated disk accesses.

The same project also studied the accident blackspots program, whose performance results are presented in Section 4.3.

7.4 Numerical and Symbolic Applications

Perhaps surprisingly, some of the most successful parallel functional applications have been numerical programs. In addition to the benefits of much higher-level coding, which include shorter, simpler (and hopefully more maintainable) code, several Sisal applications not only approach the speed of slow imperative implementations such as C, but exceed the performance produced by the fastest Fortran compilers. For parallel code, this is usually achieved without requiring any changes to the source code. Similar, though slightly less spectacular, results have been achieved for the NESL language [Ble93], mainly for generic problems such as the n-body problem [BN97]. Other generic numerical problems that have been studied in a parallel functional context include conjugate gradient algorithms [YA93, GMZ94] and various Eigen-Solver implementations [SB94, BH95].

This section surveys the most significant parallel numerical applications that have been written in these and other languages.

The Australian weather system

The Australian weather prediction model is a 10,000 line Fortran program for short-term (36-hour) weather forecasting [Les85]. Egan has re-implemented the kernel of this application as a 500-line Sisal program [Ega93] that can be called from the original Fortran shell. No significant restructuring of the code was performed, however. The parallelising Fortran compiler for the Cray-90 was unable to locate any parallelism within this subroutine.

For the Sisal version, Egan achieved a speedup of 3.7 on a 4-processor Cray-90. This represented a performance improvement of 34% over the sequential Fortran code. Subsequent work on the compiler has improved the performance of Sisal relative to Fortran, to the extent that it is now possible to achieve a relative speedup of 6.1 on an 8-processor Cray Y-MP/864 (20 iterations), representing a speedup of 5.8 over the equivalent Fortran program running on a single processor [LAN98]. The final Sisal program comprises 33 source modules – a significantly large program by most standards.

Photon Transport

The 750-line Id program `Gamteb` was written by researchers from Los Alamos National Laboratories to simulate the trajectory of photons through a carbon rod that has been divided into a number of cells of a given geometry. Each photon can be tested independently exploiting data parallelism. On the 8-processor prototype Monsoon dataflow machine, this highly-parallel application achieved a speedup of 7.4 for a problem containing 40,000 particles [HCAA93].

The same application has been written in Sisal [HLB95, HB97], but the speedups achieved on a 4-processor shared-memory Sun were not significant (1.9 relative, 1.3 absolute for 50,000 particles). The overall performance was also significantly less than for C – sequential C was 8.8 times faster than the one-processor parallel Sisal program. The poor performance is perhaps due to inefficiencies creating large intermediate data structures.

Nucleic Acids

Feeley *et al.* have worked on a parallel application for determining the three-dimensional structure of nucleic acids [FTL94]. This application involves solving a set of constraints that collectively define all legal 3D structures that can be built from the input set of nucleotides.

Each nucleotide contains one free variable describing its three-dimensional position relative to other nucleotides. This position constrains the placement of other nucleotides in the structure. The parallel implementation of the algorithm involves checking each possible solution for a nucleotide's position in parallel. The application is written as a 3500-line MultiLisp program and uses lazy task creation [MKH91, Ito96] to introduce parallel tasks.

This application has been tested on two interesting data sets. For the larger of the two data sets, `pseudoknot`, it is possible to achieve a maximum absolute speedup of 13.7 on 24 processors. This represents the limit of parallelism — additional processors result in lower speedups due to added contention. While the parallel overhead is a quite reasonable 21%, the single-processor parallel case is still 2.4 times slower than sequential C. The smaller data set, `anticodon` displays good absolute speedup of 49 on 64 processors.

Fluid Dynamics

A second large application that was developed as part of the FLARE project was the Swansea computational fluid dynamics program [RW95, GSWZ95]. In its sequential incarnation, this 2000-line program made heavy use of arrays. In order to produce a parallel implementation, *quadtree* and *trie* data structures were used instead to yield a straightforward parallel decomposition of the problem domain.

Overall, the absolute speedup achieved by this application was 2.3 on a 4-processor GRIP. Additional processors gave slight performance improvements, up to a factor of 3 on 17 processors, but gave much worse processor utilisation. This was in sharp contrast to idealised simulated results, which showed available parallelism of up to 100 simultaneous tasks. The discrepancy is probably best explained by tight data dependencies introducing significant communication costs in the real implementation. This highlights the importance of providing accurate as well as idealised simulation, as we have done in the parallel workbench described above.

A further lesson obtained from this application was the importance of providing good support for large data structures, for example *distributed applicative arrays* [KG91]. We have not yet implemented support for such structures, so would not expect good performance for programs that made heavy use of array structures in our system.

A similar application to the Swansea program is the 1000-line Id program `simple` whose purpose is to simulate hydrodynamics and heat-conduction. On an 8-processor Monsoon, Hicks *et al.* [HCAA93] report a speedup of 6.3 for 100 iterations of a 100×100 grid of nodes containing information about position and velocity, over a series of zones with different fluid characteristics. This application has also been implemented in Sisal, where researchers achieved relative speedups of 4.3 on an 8-processor Cray Y-MP/864 and 13.9 on a 20-processor Sequent Symmetry for 62 iterations [LAN98]. In both cases the Sisal version was significantly faster than the single-processor Fortran code, representing speedups over Fortran of 4.1 and 13.7 respectively.

Tidal Prediction

Hartel *et al.* have used Miranda to produce a 560 line tidal prediction program, using skeletons to expose the parallelism in this program [HHL$^+$95]. A "communication lifting" transformation is applied in order to exploit wavefront parallelism in a grid performing computational fluid dynamics operations that involve solving partial differential equations in a data-parallel fashion. The program uses a tile-based partitioning approach similar to that we have used for the Accident Blackspots program.

The relative speedup achieved for this application is 2.5 on a 4-processor shared-memory machine, though the application would presumably scale to larger shared-memory systems if these were available, by simply introducing additional tiles. Unfortunately, this is still 58% slower than sequential C, however, and therefore considerably slower than could be expected for a Sisal implementation of this application.

Global Ocean Circulation

A similar application to the tidal prediction problem is the global ocean circulation model that has been converted to Id from the Fortran original [SAC+98]. This program has a regular control structure (the central part is a triply nested loop) but an irregular data structure. The application was tuned for parallel execution on Monsoon using loop unrolling and the introduction of k-bounded loops [AN90] for throttling excess parallelism. While no absolute speedup figures are available, performance results for realistic data-sets showed clearly that the Id application required fewer clock cycles to execute each required operation on the 8-processor Monsoon prototype than Fortran on a 128-processor CM-5.

Symbolic Computation and Computer Algebra

Schreiner has applied his small strict para-functional language `pD` to a number of problems taken from computer algebra: a linear equation solver that is similar to the one presented in Section 4.6; two programs to compute multivariate polynomial resultants; and part of a polynomial factorisation algorithm.

Highly significantly, Schreiner's performance results show that good absolute speedup can be achieved using his approach [Sch95]. Compared with sequential C, Schreiner achieved performance of 14 on a 16-processor shared-memory system for the linear equation solver (his best result). Sequential performance is also broadly in line with that obtained for the corresponding C programs. Although these applications are small, they do suggest that parallel symbolic computation is amenable to exploitation by functional programming techniques.

7.5 Digital Signal Processing

In his thesis, Reekie describes the design of a parallel digital signal processing system written using a visual dialect of Haskell [Ree95]. While no performance figures are available, the thesis is interesting in introducing a number of laws concerning functional process networks that could perhaps apply to behavioural code written using evaluation strategies, such as the applications described in this paper.

Dennis has studied a similar application in a static dataflow context [Den95], as an exercise in parallelisation. This Sisal program is the core of a system that could be used to process information obtained from a sky-scanning optical surveillance device. A series of filters work as a parallel pipeline over several input stream of values, representing data obtained by the surveillance sensors. The application is highly parallel to the extent that throttling and other load management strategies would probably be required in a real implementation. Unfortunately, the application has not yet been implemented on real parallel hardware so no performance results are available for this application either.

7.6 Telephony

Finally, while not a purely functional implementation, and differing from the goals of our research in representing a distributed implementation of a concurrent language with explicit process control for semantic modeling, Erlang [AWWV96] has produced the first commercial distributed functional applications of which we are aware [Arm96]. The Erlang applications are both "fast enough" for real commercial use and use less memory than their counterparts in C. The largest application that has so far been programmed in Erlang is the 230,000 line Mobility Server, which acts as an intelligent call routing system linked to an internal telephone exchange, and which is in widespread use. Clearly, taken with the Lolita application which we have described here, there is a strong body of evidence to show that functional languages can be used for real, complex applications.

7.7 Related Approaches to Parallelism

The FLARE Applications

The applications produced by the FLARE project [RW95] formed a direct precursor to those described here, representing the first real attempt to write a number of reasonably large applications in a purely functional language and to produce parallel implementations of those programs. Like the applications described in this paper, the applications considered in the FLARE project were drawn from a wide variety of application areas: notably a computational fluid dynamics problem, a proof assistant, text compression and a geometric modeling system. The fluid dynamics program and the proof assistant (Veritas) are described above.

The attempts to parallelise the FLARE applications motivated the use of simulation (in this case using an idealised simulator, `hbc-pp` [RW93]) as well as real-machine execution, and spurred the long-term development of evaluation strategies for more precise machine control (the FLARE applications used only the primitive `par` and `seq` annotations). They also demonstrated the limitations of the GRIP prototype in executing such large programs, and highlighted the desirability of using stock parallel machines that could be made more generally available.

Overall parallel performance results were, however, quite promising. Depending on the application type, absolute speedups of between 4 and 15 were achieved on a 16-processor GRIP.

The Dutch Parallel Programming Toolkit

The toolkit developed as part of the Dutch Parallel Machine Project [BvH+87, HHL+95] takes an approach to parallel program development that is similar to the one we have described in this paper. As in our approach, the Dutch system provides both an interpreter and a compiler for sequential algorithmic debugging and initial overall performance optimisation, together with both simulated and real parallel machine implementations for parallel performance optimisation. The simulator supports three levels of detail: task-level, instruction-level and bus-cycle simulation. Like the GranSim simulator, the instruction-level simulation is acceptably accurate, delivering predictions that are 15%–23% too optimistic, though. The system has been used to develop the 560 line tidal prediction program discussed earlier.

Finally, it is worth noting that the compiler used in this project, FAST/FCG, has limited support for code optimisation. GHC provides many more optimisations, as well as source-level profiling (both sequential and parallel) through the use of cost-centre profiles [SP97]. These benefits are of great significance for large parallel programs.

7.8 Summary

This section has surveyed a variety of large-scale parallel functional applications mainly representing symbolic applications with irregular parallelism and written in many languages. These applications cover a wide range of programming domains from data-intensive applications such as database transaction managers to high-performance numerical calculations such as weather prediction systems or determining the structure of chemical compounds. Many applications have demonstrated that good relative speedups can be achieved, and several, notably those written in strict languages such as Sisal, MultiLisp and pD, have shown that the performance of conventional imperative languages such as C or Fortran can be exceeded with minimal programmer effort. The distributed language Erlang has shown that distributed functional applications can achieve commercial success, eclipsing their imperative counterparts through ease of construction and overall performance. These are positive and encouraging results for the work that we are undertaking.

8 Conclusions

8.1 Summary

We have described the development of several parallel symbolic programs in Glasgow Parallel Haskell (GPH). The programs are large, cover a range of application areas, and have been measured on networks of workstations, and a shared-memory multiprocessor. From our experiences with developing these applications we draw conclusions on the applications, the programming language, and the programming environment.

On the applications level the most significant result is that we are able to achieve modest wall-clock speedups over the optimised sequential versions for all but one of the programs, despite the fact that some of the programs were not written with the intention of being parallelised (see Table 5 in Section 5.1). We find that it is easy to use different parallel programming paradigms in GPH, and even to combine the paradigms within a single program.

On the language level we have been able to evaluate some long-standing claims about parallel functional programming. Both the determinism of the language proves helpful, as does the largely-implicit nature of the parallelism. Our new parallel programming technique, evaluation strategies, has been proven successful on a large scale. Particularly important for large programs we find that strategies allow a high level of abstraction to be maintained. There are two aspects to this abstraction: we can describe top-level parallelism, and also preserve module abstraction by describing parallelism over the data structures provided at the module interface ("data-oriented parallelism"). The benefits of this approach are elaborated in more detail via developing several versions of parallel programs in the PhD thesis [Loi98][Chapter 4].

On the programming environment level we have shown the importance of an integrated parallel programming environment, with facilities for prototyping parallel code, optimising the program, and visualising parallel behaviour. Although not the focus of this paper, the GPH programming environment has been developed alongside the programs, and is still being extended as detailed below.

Our motto in exploiting parallelism in large applications is "low pain, moderate gain." The goal of this approach is to bring the power of parallel processing, increasingly offered by the latest generation of desktop machines, to non-specialists in parallel programming. To achieve such "desktop parallelism" we use a programming model offering largely-implicit parallelism, namely parallel functional programming. However, our model is not restricted to machines with modest parallelism, and indeed it is possible to specify details of the parallel computation if necessary. Overall, our approach differs significantly from that usually used in the supercomputing area. In the latter it is feasible to spend a lot of effort in parallelising a program and the parallelisation is usually done by a specialist in parallel processing. With the applications presented in this paper we also hope to have demonstrated the merit of such a "desktop parallelism" approach in order to make the power of parallel processing more easily available to programmers.

8.2 Future Work

We are extending the work in several directions. Even with the existing suite of profiling and visualisation tools available it is hard to fully understand the parallel behaviour, of large irregularly-parallel programs. Additional tools are under construction and the most significant of these are as follows. The GRANCC profiler attributes the work done by a thread to a cost centre, i.e. an expression in the program [HLT97]. The strategic profiler attributes a thread to the strategy that induced it. A standard format for profiling data is being designed, and the tools may be offered in a user-friendly environment. We have experimented with a number of different ways of visualising the execution of parallel functional programs. We find that some are far more useful than others, and we intend to describe our experiences with the profilers and visualisations.

It would be useful to reason more formally about the strategies used in our programs. For example to demonstrate that two strategies are equivalent w.r.t. the amount of parallelism they generate, or that one generates more parallelism than another. We currently reason informally, and have some identities that we believe to be true, e.g. that `(par x)` is idempotent. We would like to develop a model for the threads created by a strategy, and use it to prove identities about strategies. The strategic identities can then be used to prove equalities, and inequalities between strategic functions.

We intend to improve and extend the GUM runtime system, and to port it to new platforms. Many aspects of GUM could be improved, including the work-stealing algorithm and the message-processing as suggested by measurements in [LH96b]. There are a number of obvious extensions to GUM, e.g. to introduce thread migration, i.e. the relocation of a running thread from one processor to another. A number of GUM ports are under way or planned, including to a Fujitsu AP1000, a Fujitsu AP3000, and a Beowulf platform.

In the longer-term, we would like to develop an even more implicitly-parallel language. One means of doing so would be to automatically insert strategies into a program, guided by static analyses of the program text. Strictness analysis [BHA86] indicates when it is *safe* to introduce parallelism, and granularity analysis [LH96a] indicates when it is worthwhile to do so. Because strategies are part of GpH it is then possible for the programmer to tune the parallel performance by refining the automatically generated strategies.

References

[AHPT93] G. Akerholt, K. Hammond, S.L. Peyton Jones, and P. Trinder. Processing Transactions on GRIP. In *PARLE'93 — Parallel Languages and Architectures Europe*, Munich, Germany, June 14–18, 1993. **URL:** ftp://ftp.dcs.gla.ac.uk/pub/glasgow-fp/papers/grip-transactions.ps.Z.

[Arm96] J. Armstrong. Erlang – a Survey of the Language and its Industrial Applications. In *INAP'96 — Exhibitions and Symposium on Industrial Applications of Prolog*, pp. 16–18, Hino, Tokyo, Japan, October 1996.

[AWWV96] J. Armstrong, M. Williams, C. Wikstrom, and R. Virding. *Concurrent Programming in Erlang*. Prentice-Hall, 1996.

[AN90] Arvind and R.S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Transactions on Computers*, 39(3), March 1990.

[BvH+87] H.P. Barendregt, M.C.J.D. van Eekelen, P.H. Hartel, L.O. Hertzberger, M.J. Plasmeijer, and W.G. Vree. The Dutch Parallel Reduction Machine Project. *Future Generation Computer Systems*, 3:261–270, December 1987. **URL:** ftp://ftp.cs.kun.nl/pub/CSI/SoftwEng.FunctLang/papers/barh87-PRMprojekt.ps.gz.

[Ble93] G.E. Blelloch. NESL: A Nested Data-Parallel Language (Version 2.6). Technical Report CMU-CS-93-129, Carnegie-Mellon University, April 1993.

[Ble96] G.E. Blelloch. Programming Parallel Algorithms. *Communications of the ACM*, 39(3):85–97, March 1996. **URL:** http://www.cs.cmu.edu/~scandal/cacm.html.

[BN97] G. Blelloch and G. Narlikar. A Practical Comparison of N-Body Algorithms. In *Parallel Algorithms*, Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1997. **URL:** http://www.cs.cmu.edu/afs/cs.cmu.edu/project/scandal/public/papers/dimacs-nbody.ps.gz.

[BF94] D. Boucher and M. Feeley. Construction Parallèle de l'Automate LR(0): Une Application de MultiLisp à la Compilation. In *6ième Rencontres Francophones du Parallélisme*, Université de Montréal, June 1994.

[BH95] W. Böhm and B. Hiromoto. Functional Implementations of the Jacobi Eigen-Solver. In *HPFC'95 — High Performance Functional Computing*, Denver, Colorado, April 10–12, 1995. **URL:** http://www.cs.colostate.edu/~dataflow/papers/sp95.ps.gz.

[Bra94] T.A. Bratvold. *Skeleton-Based Parallelisation of Functional Programs.* PhD thesis, Department of Computing and Electrical Engineering, Heriot-Watt University, Edinburgh, November 1994. **URL:** ftp://ftp.cee.hw.ac.uk/pub/funcprog/tab.phd.ps.Z.

[BHA86] G.L. Burn, C. Hankin, and S. Abramsky. Strictness Analysis for Higher Order Functions. *Science of Computer Programming*, 7:249–278, November 1986.

[BR94] F.W. Burton and V.J. Rayward Smith. Worst Case Scheduling for Parallel Functional Programming. *Journal of Functional Programming*, 4(1):65–75, January 1994.

[Can92] D. Cann. Retire Fortran? A Debate Rekindled. *Communications of the ACM*, 35(8):81–89, August 1992.

[Cla93] S. Clayman. *Developing and Measuring Parallel Rule-Based Systems in a Functional Programming Environment.* PhD thesis, Department of Computer Science, University College London, 1993.

[Dav96] K. Davis. MPP Parallel Haskell. In *IFL'96 — International Workshop on the Implementation of Functional Languages*, pp. 49–54, Bad Godesberg, Germany, September 1996. Draft Proceedings.

[Den95] J. Dennis. Static Mapping of Functional Programs: an Example in Signal Processing. In *HPFC'95 — High Performance Functional Computing*, pp. 149–163, Denver, Colorado, April 10–12, 1995.

[DGF97] D. Dinapoli, M. Giordano, and M.M. Furnari. A PVM-Based Distributed Parallel Symbolic System. *Journal of Advances in Engineering Software*, 28(5):303–312, 1997.

[Ega93] G. Egan. Implementing the Kernel of the Australian Weather Prediction Model in Sisal. In *SISAL'93*, pp. 11–17, San Diego, CA, USA, October 1993.

[FTL94] M. Feeley, M. Turcotte, and G. LaPalme. Using MultiLisp for Solving Constraint Satisfaction Problems: an Application to Nucleic Acid 3D Structure Determination. *Lisp and Symbolic Computation*, 7:231–247, 1994.

[GSWZ95] P.W. Grant, J.A. Sharp, M.F. Webster, and X. Zhang. Experiences of Parallelizing Finite-Element Problems in a Functional Style. *Software – Practice and Experience*, 25(9):947–974, September 1995.

[GMZ94] K.D. Gremban, G.L. Miller, and M. Zagha. Performance Evaluation of a New Parallel Preconditioner. Technical Report CMU-CS-94-205, School of Computer Science, Carnegie Mellon University, October 1994. **URL:** http://www.cs.cmu.edu/afs/cs.cmu.edu/project/scandal/public/papers/CMU-CS-94-205.ps.gz.

[HB97] J. Hammes and W. Böhm. On the Performance of Functional Programming Languages on Realistic Benchmarks. In *PDPTA'97 — International Conference on Parallel and Distributed Processing Techniques*, 1997. **URL:** http://www.cs.colostate.edu/~hammes/documents/final1.ps.Z.

[HLB95] J. Hammes, O. Lubeck, and W. Böhm. Comparing Id and Haskell in a Monte Carlo photon transport code. *Journal of Functional Programming*, 5(3):283–316, July 1995.

[Ham94] K. Hammond. Parallel Functional Programming: An Introduction. In *PASCO'94 — International Symposium on Parallel Symbolic Computation*, volume 5 of *Lecture Notes Series on Computing*, pp. 181–193, Hagenberg/Linz, Austria, 26–28 September, 1994. World Scientific. **URL:** http://www.dcs.st-and.ac.uk/~kh/papers/pasco94/pasco94.html.

[HLP95] K. Hammond, H-W. Loidl, and A. Partridge. Visualising Granularity in Parallel Programs: A Graphical Winnowing System for Haskell. In *HPFC'95 — High Performance Functional Computing*, pp. 208–221, Denver, Colorado, April 10–12, 1995. **URL:** http://www.dcs.st-and.ac.uk/~kh/papers/hpfc95/hpfc95.html.

[HLT97] K. Hammond, H-W. Loidl, and P.W. Trinder. Parallel Cost Centre Profiling. In *Glasgow Workshop on Functional Programming*, Ullapool, Scotland, September 15–17, 1997.

[HM98] K. Hammond and G. Michaelson, editors. *Research Directions in Parallel Functional Programming*. 1998. To appear.

[HMP94] K. Hammond, J.S. Mattson Jr., and S.L. Peyton Jones. Automatic Spark Strategies and Granularity for a Parallel Functional Language Reducer. In *CONPAR'94 — Conference on Algorithms and Hardware for Parallel Processing*, LNCS 854, pp. 521–532, Linz, Austria, September 6–8, 1994. Springer-Verlag. **URL:** ftp://ftp.dcs.glasgow.ac.uk/pub/glasgow-fp/papers/.

[HP92] K. Hammond and S.L. Peyton Jones. Profiling Scheduling Strategies on the GRIP Multiprocessor. In *IFL'92 — International Workshop on the Parallel Implementation of Functional Languages*, pp. 73–98, RWTH Aachen, Germany, September 1992. **URL:** ftp://ftp.dcs.gla.ac.uk/pub/glasgow-fp/papers/grip-scheduling.ps.gz.

[HHL+95] P.H. Hartel, R.F.H. Hofman, K.G. Langendoen, H.L. Muller, W.G. Vree, and L.O. Hertzberger. A Toolkit for Parallel Functional Programming. *Concurrency — Practice and Experience*, 7(8):765–793, December 1995. **URL:** ftp://ftp.fwi.uva.nl/pub/computer-systems/functional/reports/CPE_toolkit.ps.Z.

[HN91] M. Heytens and R.S. Nikhil. List Comprehensions in Agna, a Parallel, Persistent Object System. In *FPCA'91 — Conference on Functional Programming Languages and Computer Architectures*, LNCS 523, pp. 569–591, Harvard, MA, USA, 1991. Springer-Verlag.

[HCAA93] J. Hicks, D. Chiou, B.S. Ang, and Arvind. Performance Studies of the Monsoon Dataflow Processor. *Journal of Parallel and Distributed Computing*, July 1993. **URL:** http://www.csg.lcs.mit.edu:8001/monsoon/monsoon-performance/monsoon-performance.html.

[HL94] H. Hong and H-W. Loidl. Parallel Computation of Modular Multivariate Polynomial Resultants on a Shared Memory Machine. In *CONPAR'94 — Conference on Parallel and Vector Processing*, LNCS 854, pp. 325–336, Linz, Austria, September 6–8, 1994. Springer-Verlag. **URL:** http://www.dcs.gla.ac.uk/~hwloidl/publications/resultant.ps.gz.

[Hug89] R.J.M. Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2):98–107, April 1989. **URL:** http://www.cs.chalmers.se/~rjmh/Papers/whyfp.ps.

[Ito96] T. Ito. Efficient Evaluation Strategies for Structured Concurrency Constructs in Parallel Scheme Systems. LNCS 1068, pp. 22–52. Springer-Verlag, 1996.

[JSC96] Special Issue on Parallel Symbolic Computation of the *Journal of Symbolic Computation*, 21(4/6), April/June 1996. Academic Press.

[JDH97] S. Junaidu, A. Davie, and K. Hammond. Naira: A Parallel[2] Haskell Compiler. In *IFL'97 — International Workshop on the Implementation of Functional Languages*, LNCS 1467, pp. 215–231, September 10–12, St. Andrews, Scotland, 1997. Springer-Verlag.

[Jun98] S. Junaidu. *A Parallel Functional Language Compiler for Message Passing Multicomputers.* PhD thesis, School of Mathematical and Computational Sciences, University of St. Andrews, March 1998.

[Kel89] P.H.J. Kelly. *Functional Programming for Loosely-Coupled Multiprocessors.* Research Monographs in Parallel and Distributed Computing. MIT Press, 1989.

[Ken94] J.R. Kennaway. A Conflict Between Call-by-Need Computation and Parallelism. In *Workshop on Conditional Term Rewriting Systems*, LNCS 968, pp. 247–261, Jerusalem, Israel, 1994. Springer-Verlag. **URL:** ftp://ftp.sys.uea.ac.uk/pub/kennaway/publications/rootseq.ps.Z.

[Kes95] Marco Kesseler. Constructing Skeletons in Clean: The Bare Bones. In *HPFC'95 — High Performance Functional Computing*, pp. 182–192, Denver, Colorado, April 10–12, 1995. **URL:** ftp://sisal.llnl.gov/pub/hpfc/papers95/paper30.ps.

[Kes96] M. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distributed Memory.* PhD thesis, University of Nijmegen, 1996.

[KG91] H. Kuchen and G. Geiler. Distributed Applicative Arrays. Technical Report AIB 91-5, RWTH Aachen, 1991.

[LAN98] Sisal Performance Data. WWW page, June 1998. **URL:** http://www.llnl.gov/sisal/PerformanceData.html.

[Lau82] M. Lauer. *Computing by Homomorphic Images*, in B. Buchberger, G. Collins, R. Loos & R. Albrecht, editors, Computer Algebra — Symbolic and Algebraic Computation, pp. 139–168. Springer-Verlag, 1982.

[Les85] L.M. Leslie et al. A High Resolution Primitive Equations NWP Model for Operations and Research. *Australian Metereological Magazine*, 33:11–35, 1985.

[Loi98] H-W. Loidl. *Granularity in Large-Scale Parallel Functional Programming.* PhD thesis, Department of Computing Science, University of Glasgow, March 1998. **URL:** http://www.dcs.gla.ac.uk/~hwloidl/publications/PhD.ps.gz.

[Loi97] H-W. Loidl. LinSolv: a Case Study in Strategic Parallelism. In *Glasgow Workshop on Functional Programming*, Ullapool, Scotland, September 15–17, 1997. **URL:** http://www.dcs.gla.ac.uk/~hwloidl/publications/LinSolv.ps.gz.

[LH95] H-W. Loidl and K. Hammond. On the Granularity of Divide-and-Conquer Parallelism. In *Glasgow Workshop on Functional Programming*, Workshops in Computing, Ullapool, Scotland, July 8–10, 1995. Springer-Verlag. **URL:** http://www.dcs.gla.ac.uk/~hwloidl/publications/GlaFp95.ps.gz.

[LH96a] H-W. Loidl and K. Hammond. A Sized Time System for a Parallel Functional Language. In *Glasgow Workshop on Functional Programming*, Ullapool, Scotland, July 8–10, 1996. **URL:** http://www.dcs.glasgow.ac.uk/fp/workshops/fpw96/Loidl.ps.gz.

[LH96b] H-W. Loidl and K. Hammond. Making a Packet: Cost-Effective Communication for a Parallel Graph Reducer. In *IFL'96 — International Workshop on the Implementation of Functional Languages*, LNCS 1268, pp. 184–199, Bad Godesberg, Germany, September 1996. Springer-Verlag. **URL:** http://www.dcs.gla.ac.uk/~hwloidl/publications/IFL96.ps.gz.

[LT97] H-W. Loidl and P.W. Trinder. Engineering Large Parallel Functional Programs. In *IFL'97 — International Workshop on the Implementation of Functional Languages*, LNCS 1467, pp. 179–198, University of St. Andrews, Scotland, September 10–12, 1997. Springer-Verlag. **URL:** http://www.dcs.gla.ac.uk/~hwloidl/publications/IFL97.ps.gz.

[LMT+97] H-W. Loidl, R. Morgan, P.W. Trinder, S. Poria, C. Cooper, S.L. Peyton Jones, and R. Garigliano. Parallelising a Large Functional Program; Or: Keeping LOLITA Busy. In *IFL'97 — International Workshop on the Implementation of Functional Languages*, LNCS 1467, pp. 199–214, University of St. Andrews, Scotland, September 10–12, 1997. Springer-Verlag. **URL:** http://www.dcs.gla.ac.uk/~hwloidl/publications/IFL97.ps.gz.

[LL93] C. Limongelli and H-W. Loidl. Rational Number Arithmetic by Parallel P-adic Algorithms. In *ACPC'93 — Parallel Computation — Second International ACPC Conference*, LNCS 734, pp. 72–86, Gmunden, Austria, October 4–6, 1993. Springer-Verlag. **URL:** http://www.dcs.gla.ac.uk/~hwloidl/publications/p-adic.ps.gz.

[Lip71] J. D. Lipson. Chinese Remainder and Interpolation Algorithms. In *SYMSAM'71 — Symposium on Symbolic and Algebraic Manipulation*, pp. 372–391. Academic Press, 1971.

[LP95] J. Launchbury and S.L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–342, December 1995.

[MKH91] E. Mohr, D.A. Kranz, and R.H. Halstead Jr. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991. **URL:** ftp://cag.lcs.mit.edu/pub/papers/futures.ps.Z.

[MS95] G. Michaelson and N. Scaife. Prototyping a Parallel Vision System in Standard ML. *Journal of Functional Programming*, 5(3):345–382, July 1995. **URL:** ftp://ftp.cee.hw.ac.uk/pub/funcprog/ms.jfp95.ps.Z.

[MT93] S. Mitrovic and M. Trobina. Computer Vision Algorithms in Sisal. In *SISAL'93*, pp. 114–119, San Diego, CA, USA, October 1993.

[MSS94] R.G. Morgan, M.H. Smith, and S. Short. Translation by Meaning and Style in Lolita. In *International BCS Conference — Machine Translation Ten Years On*, Cranfield University, November 1994.

[Nik91] R.S. Nikhil. ID Reference Manual. Technical Report CSG Memo 284-2, Laboratory for Computer Science, M.I.T., July 1991. **URL:** ftp://csg-ftp.lcs.mit.edu/pub/papers/csgmemo/memo-284-2.ps.gz.

[NSvP91] E.G.J.M.H. Nöcker, J.E.W. Smetsers, M.C.J.D. van Eekelen, and M.J. Plasmeijer. Concurrent Clean. In *PARLE'91 — Parallel Architectures and Languages Europe*, LNCS 505, pp. 202–219, Veldhoven, The Netherlands, June 1991. Springer-Verlag. **URL:** ftp://ftp.cs.kun.nl/pub/CSI/SoftwEng.FunctLang/papers/noce91-concurrentclean.ps.gz.

[Pey96] S.L. Peyton Jones. Compiling Haskell by Program Transformation: a Report from the Trenches. In *ESOP'96 — European Symposium on Programming*, LNCS 1058, pp. 18–44, Linköping, Sweden, April 22–24, 1996. Springer-Verlag. **URL:** http://www.dcs.gla.ac.uk/fp/authors/Simon_Peyton_Jones/comp-by-trans.ps.gz.

[PHA+97] J.C. Peterson, K. Hammond, L. Augustsson, B. Boutel, F.W. Burton, J.H. Fasel, A.D. Gordon, R.J.M. Hughes, P. Hudak, T. Johnsson, M.P. Jones, S.L. Peyton Jones, A. Reid, and P.L. Wadler. *Report on the Non-Strict Functional Language, Haskell, Version 1.4*, 1997.

[PS93] A.P. Poulovassilis and C. Small. A Domain-Theoretic Approach to Logic and Functional Databases. In *VLDB'93 — International Conference on Very Large Databases*, pp. 415–426, 1993.

[Ree95] H.J. Reekie. *Realtime Signal Processing: Dataflow, Visual and Functional Programming*. PhD thesis, School of Electrical Engineering, University of Technology at Sydney, 1995.

[Roe91] P. Roe. *Parallel Programming using Functional Languages*. PhD thesis, Department of Computing Science, University of Glasgow, February 1991.

[RW93] C. Runciman and D. Wakeling. Profiling Parallel Functional Computations (without Parallel Machines). In *Glasgow Workshop on Functional Programming*, Workshops in Computing, pp. 236–251, Ayr, Scotland, July 5–7, 1993. Springer-Verlag.

[RW95] C. Runciman and D. Wakeling. *Applications of Functional Programming*. UCL Press, 1995.

[SP95] P.M. Sansom and S.L. Peyton Jones. Time and Space Profiling for Non-Strict Higher-Order Functional Languages. In *POPL'95 — Symposium on Principles of Programming Languages*, San Francisco, California, January 1995. ACM Press. **URL:** ftp://ftp.dcs.glasgow.ac.uk/pub/glasgow-fp/papers/profiling.ps.Z.

[SP97] P.M. Sansom and S.L. Peyton Jones. Formally Based Profiling for Higher-Order Functional Languages. *ACM Transactions on Programming Languages and Systems*, 19(2):334–385, March 1997. **URL:** http://www.dcs.gla.ac.uk/fp/authors/Patrick_Sansom/1997_profiling_TOPLAS.ps.gz.

[Sch95] W. Schreiner. Application of a Para-Functional Language to Problems in Computer Algebra. In *HPFC'95 — High Performance Functional Computing*, pp. 182–192, Denver, Colorado, April 10–12, 1995.

[Sha98] A. Shaw. Impala Application Suite. WWW page, January 1998. **URL:** http://www.csg.lcs.mit.edu:8001/impala/.

[SAC+98] A. Shaw, Arvind, K.-C. Cho, C. Hill, R.P. Johnson, and J. Marshall. A Comparison of Implicitly Parallel Multithreaded and Data Parallel Implementations of an Ocean Model. *Journal of Parallel and Distributed Computing*, 48(1):1–51, January 1998.

[SMT+95] E. Smirni, A. Merlo, D. Tessera, G. Haring, and G. Kotsis. Modeling Speedup of SPMD Applications on the Intel Paragon: a Case Study. In *HPCN'95 — High Performance Computing and Networking*, LNCS 919, pp. 94–101, Milan, Italy, May 1995. Springer-Verlag.

[SB95] A.C. Sodan and H. Bock. Extracting Characteristics from Functional Programs for Mapping to Massively Parallel Machines. In *HPFC'95 — High Performance Functional Computing*, pp. 134–148, Denver, Colorado, April 10–12, 1995. **URL:** ftp://sisal.llnl.gov/pub/hpfc/papers95/paper14.ps.

[Ske91] S. Skedzielewski. *Parallel Functional Languages and Compilers*, chapter Sisal. Frontier Series. ACM Press, 1991.

[SB94] S. Sur and W. Böhm. Analysis of Non-Strict Functional Implementations of the Dongarra-Sorensen Eigensolver. In *ICS'94 — International Conference on Supercomputing*, Manchester, U.K., June 1994. **URL:** http://www.cs.colostate.edu/~dataflow/papers/ics94b.ps.gz.

[Tar75] R.E. Tarjan. Efficiency of a Good, but not Linear Set Union Algorithm. *Journal of the ACM*, 22:215–225, 1975.

[Tay97] F.S. Taylor. *Parallel Functional Programming by Partitioning*. PhD thesis, Department of Computing, Imperial College, London, 1997.

[TG95] G. Tremblay and G.R. Gao. The impact of laziness on parallelism and the limits of strictness analysis. In *HPFC'95 — High Performance Functional Computing*, pp. 119–133, Denver, Colorado, April 10–12, 1995. **URL:** ftp://sisal.llnl.gov/pub/hpfc/papers95/paper11.ps.

[TBD+98] P.W. Trinder, E. Barry Jr., M.K. Davis, K. Hammond, S.B. Junaidu, U. Klusik, H-W. Loidl, and S.L. Peyton Jones. GPH: An Architecture-Independent Functional Language. *IEEE Transactions on Software Engineering*, 1998. Submitted for publication.

[THLP98] P.W. Trinder, K. Hammond, H-W. Loidl, and S.L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, January 1998. **URL:** http://www.dcs.glasgow.ac.uk/~hwloidl/publications/strategies.ps.gz.

[THM+96] P. Trinder, K. Hammond, J.S. Mattson Jr., A.S Partridge, and S.L. Peyton Jones. GUM: a Portable Parallel Implementation of Haskell. In *PLDI'96 — Programming Languages Design and Implementation*, pp. 79–88, Philadelphia, PA, USA, May 1996. **URL:** ftp://ftp.dcs.glasgow.ac.uk/pub/glasgow-fp/authors/Philip_Trinder/gumFinal.ps.Z.

[Win97] N. Winstanley. A Type-Sensitive Preprocessor for Haskell. In *Glasgow Workshop on Functional Programming*, Ullapool, Scotland, September 15–17, 1997. **URL:** http://www.dcs.gla.ac.uk/~nww/Papers/GlaFP.ps.Z.

[WH96] J. Wu and L. Harbird. A Functional Database System for Road Accident Analysis. *Advances in Engineering Software*, 26(1):29–43, 1996.

[YA93] D. Yeung and A. Agarwal. Experience with Fine-Grain Synchronization in MIMD Machines for Preconditioned Conjugate Gradient. In *PPoPP'93 — Symposium on Principles and Practice of Parallel Programming*, May 1993. **URL:** ftp://cag.lcs.mit.edu/pub/papers/fine-grain.ps.Z.