# Engineering Large Parallel Functional Programs

Hans-Wolfgang Loidl[1] and Phil Trinder[2]

[1] Department of Computing Science, University of Glasgow, Glasgow, Scotland, U.K.
E-mail: hwloidl@dcs.glasgow.ac.uk
[2] The Computing Department, The Open University, Milton Keynes, England, U.K.
E-mail: P.W.Trinder@open.ac.uk

**Abstract.** The design and implementation of useful programming languages, whether sequential or parallel, should be driven by large, realistic applications. In constructing several medium- and large-scale programs in Glasgow Parallel Haskell, GpH, a parallel extension of Haskell, the group at Glasgow has investigated several important engineering issues:

– *Real Application Parallelism.* The programs achieve good wall-clock speedups and acceptable scale-up on both a shared-memory and a distributed memory machine. The programs typify a number of application areas and use a number of different parallel paradigms, e.g. pipelining or divide-and-conquer, often combining several paradigms in a single program.
– *Language Issues.* Although the largely implicit parallelism in GpH is a satisfactory programming model in general the base constructs for introducing and controlling parallelism tend to obfuscate the semantics of large programs. As a result we developed evaluation strategies, a more abstract, and systematic mechanism for introducing and controlling parallelism.
– *Engineering Environment.* The development and performance tuning of these programs emphasised the importance of an integrated engineering environment. In the process we have refined components of this environment like the simulator, the runtime system, and the profiling tools.

## 1 Introduction

The development and especially the performance tuning of a parallel lazy algorithm requires more than just a compiler that incorporates parallelism directives for a parallel runtime-system. In the absence of a system for implicit parallelism, the programmer needs some form of language support for exposing parallelism and tools for examining the dynamic behaviour of the execution. These aspects become painfully clear when the program to be parallelised is fairly large.

We have recently constructed an integrated programming environment to support programming in Glasgow Parallel Haskell, GpH, a parallel extension of Haskell. The environment consists of a highly parameterised simulator, GranSim, as well as a portable parallel runtime-system, GUM. This environment enables the programmer to develop a parallel program in an idealised setting of a simulator before focusing on machine specific aspects. A set of visualisation tools, available for both systems, shows activity, granularity, and resource management on several levels of detail. Both GranSim and GUM are built on top of the Glasgow Haskell Compiler, GHC, using its support for program analysis, optimisation, and profiling.

The reasons for undertaking the time consuming effort of parallelising medium- to large-scale programs are threefold:

- The programs allow us to investigate whether GpH can achieve wall-clock speedups and acceptable scale-up for a range of real applications on both distributed- and shared-memory machines.
- Large programs are essential to evaluating the strengths and weaknesses of any language. This is especially true of functional languages whose modularity and high level of abstraction help to structure large programs. In addition to the aspects of engineering sequential applications, we investigate the language constructs for introducing and controlling parallelism.
- Large programs also test our implementation technology and tools. They reveal whether our implementation techniques scale-up to handle large bodies of code running for extended periods. They also test whether our profiling and visualisation tools provide us with the information required to tune the parallel behaviour.

Most of the programs we have studied entail symbolic computation with functions operating on complex data structures. These programs are suitable because they make greater use of the strengths of functional languages like algebraic data types and higher-order functions. Our hope is to parallelise the program with limited effort, and certainly without a total rewrite. This approach is made possible because a non-strict functional language does not enforce a specific evaluation order and is in stark contrast to the trend in parallel (super-) computing where considerable programming time is invested in every program.

We have written many small GpH programs, several medium-sized programs and parallelised one large program, the Lolita natural language engineering system, which is described in detail in [7]. These programs use various different parallelism paradigms, mostly fine-grained parallelism, and in some cases a limited, benign form of speculative parallelism. In this paper we discuss the following programs:

- *Alpha-Beta search*, a program for performing a heuristic search in a tree structure, usually used in game programming. It is a typical program for AI applications.
- *Accident Blackspots*, a program for locating accident blackspots from police traffic accident reports. It is typical of data-intensive complex-query programs.
- *LinSolv*, a program for finding an exact solution of a system of linear equations. It uses a structure typical to many algorithms in computer algebra.

The structure of the remainder of the paper is as follows. Section 2 describes both parallelism in GpH, and evaluation strategies. Section 3 discusses the guidelines we are developing for parallelising large functional programs. Section 4 describes several parallel variants of the Alpha-Beta search algorithm. Section 5 describes the parallelisation of the accident blackspots program, including obtaining wall-clock speedups. Section 6 outlines the parallelisation of the LinSolv linear equation solver. Section 7 discusses what has been learnt from engineering these programs including design lessons for the parallel implementation of functional languages.

## 2 GPH and Evaluation Strategies

### 2.1 GPH Parallelism

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. In GPH we have to specify only a few aspects of the parallel execution, with the runtime system managing details like synchronisation of threads and transfer of data. The programmer is only required 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` 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* [10].

Since control of sequencing can be important in a parallel language [12], we introduce a sequential composition operator, `seq`. If `e1` is not $\perp$, the expression `e1 ‘seq‘ e2` 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`. Note that this presentation of strategies is based on Haskell 1.2.

### 2.2 Evaluation Strategies

*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 parallel 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 complete description and discussion of strategies can be found in [15].

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 nullary tuple `()`.

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

### 2.3 Strategies Controlling Evaluation Degree

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

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

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

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

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

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

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

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

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

### 2.4 Combining Strategies

Because evaluation strategies are just normal higher-order functions, they can be combined using the full power of the language, e.g. passed as parameters or composed using the function composition operator. Elements of a strategy are combined by sequential or parallel composition (`seq` or `par`). Many useful strategies are higher-order, for example, `seqList` below is a strategy that sequentially applies a strategy to every element of a list. The strategy `seqList r0` evaluates just the spine of a list, and `seqList rwhnf` evaluates every element of a list to WHNF.

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

### 2.5 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 a useful example of data-oriented parallelism; for example the `parMap` function defined below applies its function argument to every element of a list in parallel. 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
```

### 2.6 Control-oriented parallelism

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

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

demanding = flip seq
sparking  = flip par
```

Several control-oriented strategies, including an example of how to describe a thresholding mechanism as part of a strategy, are discussed in the accompanying paper [7].

## 3 Parallelisation Guidelines

From our experiences engineering several parallel programs using evaluation strategies we are developing guidelines for parallelising large non-strict functional programs. The approach is top-down, starting with the top level pipeline, and then parallelising successive components of the program. The first five stages are machine-independent. Our approach uses several ancillary tools, including time profiling [14] and the GranSim simulator [1]. Several stages use GranSim, which is fully integrated with the GUM parallel runtime system [17]. A crucial property of GranSim is that it can be parameterised to simulate both real architectures and an idealised machine with, for example, zero-cost communication and an infinite number of processors.

The stages in our methodology are as follows.

1. *Sequential implementation.* Start with a correct implementation of an inherently-parallel algorithm or algorithms.
2. *Seek top-level parallelism.* Often a program will operate over independent data items, or the program may have a number of stages, e.g. lex, parse and typecheck in a compiler. Both top-level data parallelism and pipelining are very easy to specify, and often gain some parallelism for minimal change.
3. *Time Profile* the sequential application to discover the "big eaters", i.e. the computationally intensive pipeline stages.
4. *Parallelise Big Eaters* using evaluation strategies. It is sometimes possible to introduce adequate parallelism without changing the algorithm; otherwise the algorithm may need to be revised to introduce an appropriate form of parallelism, e.g. divide-and-conquer or data-parallelism.
5. *Idealised Simulation.* Simulate the parallel execution of the program on an idealised execution model, i.e. with an infinite number of processors, no communication latency, no thread-creation costs etc. This is a "proving" step: if the program isn't parallel on an idealised machine it won't be on a real machine. This often indicates that some restructuring of the code is necessary to achieve good parallel performance. We now use GranSim, but have

previously used HBCPP [13]. A simulator is often easier to use, more heavily instrumented, and can be run in a more convenient environment, e.g. a workstation.
6. *Realistic Simulation.* GranSim can be parameterised to closely resemble the GUM runtime system for a particular machine, forming a bridge between the idealised and real machines. A major concern at this stage is to improve thread granularity so as to offset communication and thread-creation costs.
7. *Real Machine.* In an execution of the program on a real machine further aspects, such as the impact of operating system calls on the parallel performance, have to be considered. In order to analyse the parallel behaviour of the program, the GUM runtime system supports some of the GranSim performance visualisation tools. This seamless integration helps understand real parallel performance.

It is more conventional to start with a sequential program and then move almost immediately to working on the target parallel machine. This has often proved highly frustrating: the development environments on parallel machines are usually much worse than those available on sequential counterparts, and, although it is crucial to achieve good speedups, detailed performance information is frequently not available. It is also often unclear whether poor performance is due to use of algorithms that are inherently sequential, or simply artifacts of the communication system or other dynamic characteristics. In its overall structure our methodology is similar to others used for large-scale parallel functional programming [3].

## 4 Alpha-Beta Search

The first example program is the Alpha-Beta search algorithm, typical of artificial intelligence applications. It is mainly used for game-playing programs to find the best next move by generating all possible moves up to a certain depth, applying a static evaluation function to each of the leaves in this search tree, and combining the result by picking the best move for the player assuming that the opponent picks the worst move for the player.

We discuss two versions of the Alpha-Beta search algorithm: a *simple* version, and a *pruning* version. Both versions are based on the Miranda[1] code presented by John Hughes [4] in order to demonstrate the strengths of lazy functional languages. The pruning version relies on laziness to improve the efficiency of the sequential algorithm by pruning the search tree based on intermediate results. In this section we parallelise both versions 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 comparison of both algorithms with a discussion of their parallel performance is given in [8].

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

### 4.1 Simple Algorithm

In the simple algorithm each possible next move is evaluated independently yielding a compositional structure of the algorithm. The result is either the maximum (player's move) or the minimum (opponent's move) of the evaluations of all next positions. This algorithm can be very naturally described as a sequence of function compositions performing the following tasks:

1. Build a tree with positions as nodes and all possible next moves as subtrees.
2. Prune the tree, which might be infinite at this stage, to a fixed depth to bound the search.
3. Map a static evaluation function over all nodes of the tree.
4. Crop off subtrees from winning or losing positions. If such a position is found it is not necessary to search deeper in a subtree.
5. Finally, pick the maximum or minimum of the resulting evaluations in order to determine the value of the current position via `mise f g`. The functions `f` and `g` represent the combination functions for the two players and alternate when traversing the tree.

**Dynamic Behaviour.** The compositional nature of this algorithm makes parallelisation rather easy. For both versions of the algorithm we study the following three sources of parallelism:

*Parallel Static Evaluation Function.* The idea of a parallel static evaluation function is to reduce the costs of the function, which will be mapped over the leaves of the pruned search tree. In our example implementations, the static evaluation function is very simple: it computes the distance of the current position to a set of known winning positions. The parallel version computes all distances in parallel.

*Parallel Higher-order Functions over Trees.* Parallelising the definitions of some higher-order functions is a bottom-up approach. It can be used for the parallelisation of many functional programs. In this case we use a parallel version of a map function over search trees.

*Data Parallelism over All Possible Next Moves.* In a data parallel approach the goal is to evaluate all possible next moves in parallel. It is a top-down approach and turns out to be the best source of parallelism for such a compositional algorithm with no dependencies between the evaluations of the subtrees. A simple `parMap rnf` strategy can be used to capture the dynamic behaviour of this function. The only necessary change in the algorithm affects the `mise` function in Stage 5 of the algorithm, shown in Figure 1. As arguments to this function either the binary `max` or `min` function is 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)
```

**Fig. 1.** Data parallel combination function in the simple Alpha-Beta search algorithm.

**Performance Measurements.** Our measurements of both versions of the algorithm under the GranSim simulator are summarised in Table 1. In all test runs we used a GranSim setup modelling a tightly connected distributed memory machine with 32 processors, a latency of 64 machine cycles, and bulk fetching. The first four data columns of this table show the results of the simple algorithm when using the different sources of parallelism. All runtimes are given in machine-independent kilocycles. The work column measures the total work compared to a sequential run and is therefore a measure of the redundant work, in particular of speculative parallelism. The three horizontal sections in the table represent three different positions that have been analysed: a standard opening position (I), a winning position (II), and a position generating a large search tree (III).

**Table 1.** Measurements of the simple and the pruning Alpha-Beta search algorithm

| | Simple Algorithm | | | | Pruning Algorithm | | | |
|---|---|---|---|---|---|---|---|---|
| | Runtime (kilocycles) | Avg Par | Total Work | Speedup | Runtime (kilocycles) | Avg Par | Total Work | Speedup |
| *Position I* | (standard opening) | | | | | | | |
| Sequential | 60,297 | | | | 34,363 | | | (1.75) |
| Par Static Eval | 21,091 | 3.1 | 108% | 2.85 | 12,099 | 3.1 | 109% | 2.84 |
| Data Par | 3,503 | 26.4 | 153% | 17.21 | 2,265 | 23.7 | 156% | 15.17 |
| Par h.o. fcts | 4,954 | 20.9 | 172% | 12.16 | 4,248 | 24.2 | 299% | 8.08 |
| *Position II* | (early solution) | | | | | | | |
| Sequential | 4,427 | | | | 4,703 | | | (0.94) |
| Par Static Eval | 1,772 | 2.9 | 116% | 2.49 | 1,898 | 2.9 | 117% | 2.47 |
| Data Par | 1,152 | 13.9 | 362% | 3.84 | 1,075 | 13.1 | 299% | 4.37 |
| Par h.o. fcts | 759 | 9.6 | 165% | 5.83 | 811 | 9.0 | 155% | 5.79 |
| *Position III* | (large search tree) | | | | | | | |
| Sequential | 145,720 | | | | 90,377 | | | (1.61) |
| Par Static Eval | 48,808 | 3.3 | 111% | 2.98 | 29,891 | 3.3 | 109% | 3.02 |
| Data Par | 6,621 | 29.1 | 132% | 22.00 | 7,699 | 16.2 | 138% | 11.73 |
| Par h.o. fcts | 9,345 | 21.4 | 137% | 15.59 | 8,093 | 24.6 | 220% | 11.16 |

The *parallel static evaluation function* generates conservative parallelism shown by the small amount of total work performed. However, the resulting parallelism is rather small and very fine-grained, yielding a rather poor speedup. The *data parallelism* over all next positions proves to be the best source of parallelism. The simple algorithm will only cut-off subtrees if it finds a winning position in one of the subtrees. Therefore, this data parallelism is conservative except for the case where a winning position is found as in Position II. Finally, the *higher order functions* approach generates the largest amount of redundant work for Positions I and III, which is shown by the high total work percentage. Here a parallel map of the static evaluation function is used. However, this also maps the evaluation function on nodes that are actually pruned in the sequential algorithm.

### 4.2 Pruning Algorithm

The simple algorithm described in the previous section lacks one crucial optimisation of the Alpha-Beta search: the pruning of subtrees based on intermediate results. The pruning algorithm returns an increasing list (player's move) of approximations with the exact value as last list element rather than a single value. The main pruning function, `minleq`, has to test whether the opponent's move from a subtree, 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 [4].

**Dynamic Behaviour.** Unfortunately, the pruning version seriously complicates the parallelisation of the algorithm. We have already seen in the simple algorithm that the most promising source of parallelism is the parallel evaluation of all next positions. However, using a simple `parList rnf` strategy over all next positions is no longer advisable, since this might result in a lot of redundant work, if many subtrees can be pruned. The measurements of the data parallel strategy on the pruning algorithm in Table 1 show a rather high degree of redundant work. In fact, in the data parallel strategy on Position III the parallel simple version is even faster than the highly speculative pruning version of the algorithm!

A better approach for parallelisation is to force only an initial segment in the list of possible next positions. We call the length of this segment the "force length". We have experimented with static force lengths as well as dynamic force lengths that depend on the level in the search tree. To date the best results have been obtained from using a static force length as shown in the code in Figure 2. Note that the force length represents a trade-off between increasing the degree of parallelism and reducing the total amount of work being done.

```
-- 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` () )
```

**Fig. 2.** Strategy for a pruning Alpha-Beta search with a static force length

**Performance Measurements.** Figure 3 compares the speedups of the pruning version of Alpha-Beta search under GranSim in the same setup as in the previous measurements. The x-axis shows the static force length, the y-axis the speedup. The left hand graph uses a program implementing tic-tac-toe, the right hand graph uses an implementation of a similar game, escape, with a search space of comparable size but asymmetric winning conditions.

The left hand graph shows for the data parallel strategy a large improvement when increasing the force length, in particular for Position III. A purely conservative data parallel strategy (i.e. the force length is 0) achieves a speedup of only 8.58 because the amount of available parallelism drops early on in the computation. In contrast, with a force length of 4 the speedup is 15.71. After that the percentage of redundant work done in the parallel algorithm increases too much to achieve a further improvement. For Position II, which finds a winning position early on in the search, parallelism can achieve hardly any improvement because almost all potential parallelism in the algorithm is pruned. The right hand side of Figure 3 shows an even larger improvement with increasing force length. In both cases the versions additionally using a parallel static evaluation function usually outperform the versions with data parallelism alone. The small amount of conservative parallelism in the static evaluation can be usefully exploited while the machine is idle.

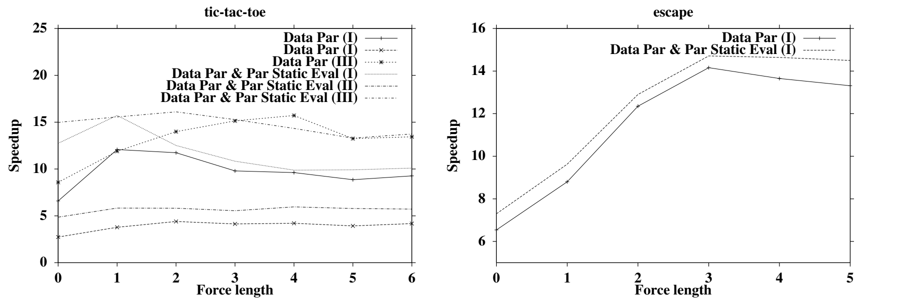


**Fig. 3.** Speedup of a pruning Alpha-Beta search with varying static force length

## 5 Accident Blackspots

This section outlines a data-intensive GPH program that solves a real problem using real data and achieves good wall-clock speedups on two very different parallel machines: a shared memory Sun SPARCserver and a distributed-memory network of workstations. A more detailed description of the program and parallelisation can be found in [16].

### 5.1 Problem Description

Given a set of police accident records (modified only 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: they may have 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 problem 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}}`.

### 5.2 Sequential Implementations

*The PFL Implementation.* The application was originally written at the Centre for Transport Studies [18] in PFL and has subsequently been rewritten in Haskell. PFL is an interpreted functional language [11], designed specifically to handle large deductive databases. Unusually for a functional language, PFL provides a uniform persistent framework for both data and program.

*The Haskell Implementation.* The Haskell implementation constructs a binary 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.

### 5.3 Parallel Variants

Following our 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. Table 2 reports the results obtained from the simulators when just 1000 accidents are partitioned, runtimes and work are in GRANSIM megacycles.

**Table 2.** Idealised simulation of Blackspots

| Parallel Variant | Work (megacycles) | Average Parallelism | Runtime (megacycles) |
|---|---|---|---|
| Pipeline only | 327 | 1.2 | 273 |
| Par. Pipeline Stages | 335 | 2.8 | 124 |
| Par. Pipeline Stages & preconstructed Ixs | 304 | 3.5 | 87 |
| Geographically Partitioned | 389 | 3.7 | 105 |

*Pipeline only.* The first version simply converts the 4 phases of the program outlined in Section 5.2 into a pipeline. The speedup of 1.2 is disappointingly 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. 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.

*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 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-grained parallelism and the volume of data being communicated.

**Table 3.** Realistic SPARCserver simulation of Blackspots

| Parallel Variant | Work (megacycles) | Average Parallelism | Runtime (megacycles) |
|---|---|---|---|
| Par. Pipeline Stages & preconstructed Ixs | 393 | 2.3 | 171 |
| Geographically Partitioned | 394 | 3.7 | 105 |

*Geographically Partitioned.* A very different, coarse-grained, parallel structure can be obtained by splitting the accident data into geographical areas, each area, or *tile*, can be partitioned in parallel before aggregating the results. Accidents occuring near the edge of a tile must be treated specially. In fact this approach is only feasible because every accident has a grid reference and we assume that accidents occuring more than 200m apart cannot be at the same site. Accidents occuring 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.

Breaking the data into tiles reduces the work required to form a partition as long as the border is much 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.

Our environment, in particular GranSim and strategies, have allowed us to carry out low-cost experiments with several possible parallel variants of the program. Most of the program, the partitioning algorithm in particular, remained substantially unchanged while different parallel strategies were investigated. The tiled variant is selected for execution on the real machine because it delivers good coarse-grained parallelism under both idealised and realistic simulation.

### 5.4 Apparatus

*Machines.* 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.

*Data.* The original data set of 7300 accident records occupies 0.3Mb and is split into 2 different sized tiles: 8 small tiles of ca. 1000 accidents occupying 37Kb, and 4 large tiles of ca. 2000 accidents occupying 73Kb. Both architectures use a shared file system, via NFS, and are warm started, i.e. the program 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.

*Program.* One change is required to the GranSim version of the program to enable it to run under GUM. GUM processors don't inherit file handles from the main thread, and hence to permit them to simultaneously read files the program uses the 'unsafe' C-interface supported by Glasgow Haskell.

### 5.5 Measurements

**Original Data Set.** For these measurements the data is in 8 small tiles on the workstations, and in 4 large tiles on the SPARCserver. The runtimes of the program under GUM drop from 136.52 seconds on 1 workstation to 76.53 seconds on 2 workstations, and to 26.74 seconds on 8 workstations. The runtime under GUM on a single processor is 28% longer than the runtime for the optimised program because GUM imposes some additional overheads [17].

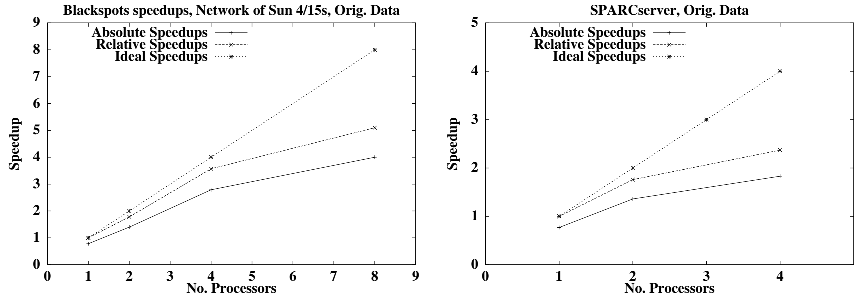


**Fig. 4.** Speedups of Blackspots on original data

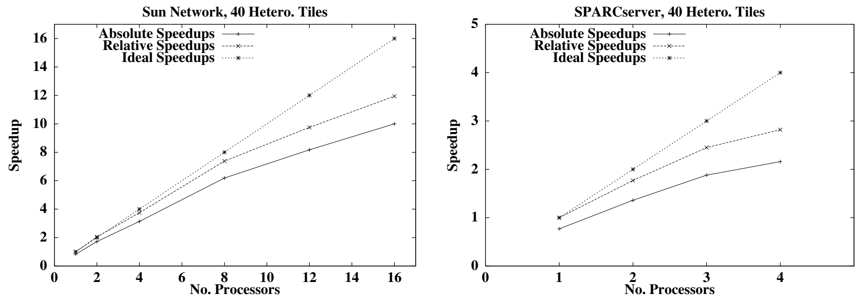


**Fig. 5.** Speedups of Blackspots on heterogeneous tiles

Parallelism results are often reported as speedup graphs like Figure 4. The top line is ideal, or linear speedup. The second line is the *relative* speedup, i.e. compared to a single processor running the parallel program. The bottom line is the *absolute* or wall-clock speedup, i.e. compared to a single processor running the optimised sequential code.

These results are the first wall-clock speedups obtained for a GpH program solving a real problem using real data. On the SPARCserver the speedup is good at 2 processors, but poor thereafter. On the workstations the speedups are good up to 4 processors (3.6 relative speedup), but fall off thereafter. This indicates that the initial data set is too small to get good results on the larger parallel machine configurations.

**Multiple Tiles per Processor.** The data set can be increased by using larger tiles or by using more tiles. The latter approach is taken mainly because it makes good use of the dynamic load management supported by GUM: when a processor becomes idle it locates the next work item, in this case a tile. With many tiles each processor can be kept busy. Furthermore, Section 5.3 showed that many small tiles are more efficient than a few large tiles.

Figure 5 shows the speedups obtained partitioning 1.8Mb of data in 40 tiles: 32 small and 8 large. The larger data set gives an improved speedup on both architectures: 12 relative and 10 absolute on 16 workstations, and 2.8 relative and 2.2 absolute on the SPARCserver.

## 6 A Linear Equation Solver

In this section we discuss LinSolv, a parallel algorithm for finding an exact solution of a system of linear equations over integer values. It represents a typical application from the area of symbolic computation dealing with arbitrary precision integer values rather than floating point numbers. A more detailed discussion of this algorithm including a comparison with the pre-strategy code is given in [9].

### 6.1 Sequential Algorithm

LinSolv uses an approach that is very common for computer algebra algorithms: a multiple homomorphic images approach [5]. The main idea of this approach is to solve a problem in a set of simpler domains, called homomorphic images, and then to reconstruct, or lift, the overall solution from the solutions in the individual domains.

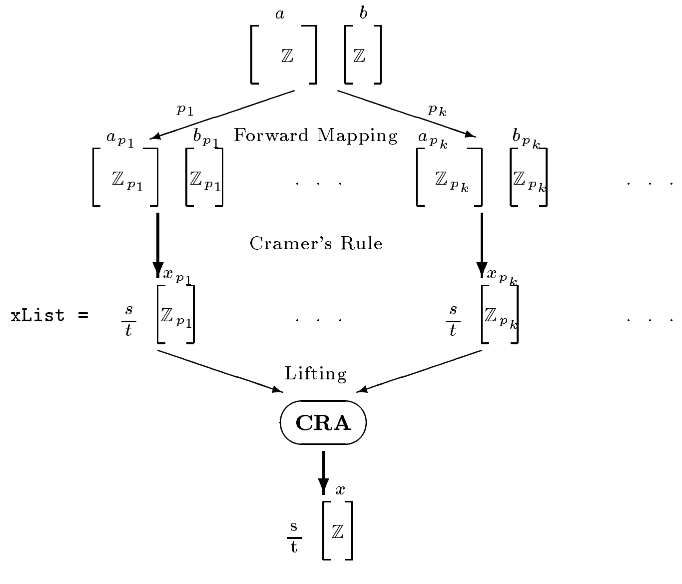


**Fig. 6.** Structure of the LinSolv algorithm

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. Even in the sequential case the gain in efficiency by using cheap arithmetic outweighs the additional costs of combining the solutions. In the case of $\mathbb{Z}$ as original domain the well-studied Chinese Remainder Algorithm (CRA) can be used in the combine step [6]. This overall structure of the algorithm is shown in Figure 6. Note that we use an algorithm that factors out the rational part $\frac{s}{t}$ in the result matrix so that the result vector $x$ only contains integer values.

In order to describe the parallelism in this algorithm it is important to understand the structure of the main intermediate data structure, `xList`, an infinite list of the solutions in all homomorphic images. Each solution is a list with the following components: a prime number, which is the basis of the homomorphic image; the image of the determinant of the input matrix, which is needed to decide whether the result in this image can be used for constructing the overall result; and the result vector of solving the linear system of equations.

### 6.2 Parallelisation

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

**Fig. 7.** Strategy for the parallel LinSolv algorithm

The strategy in Figure 7 controls the parallelism over three variables in the algorithm: `noOfPrimes`, a guess of how many prime numbers are needed in order to construct the overall result; `xList`, the infinite list of homomorphic solutions; and `x`, the result vector in the original domain. Based on the result of `noOfPrimes` the strategy uses `parListN` to generate data parallelism over an initial segment of `xList`. This is similar to the use of a static force length in the Alpha-Beta search algorithm. However, in this case the guess is conservative and will not generate redundant computation. Note that without controlling the parallelism over `xList` the CRA will demand each solution sequentially, because it performs a list fold operation. The final strategy application `parList rnf x` specifies that all elements of the overall result should be combined in parallel.

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 determinant of the original matrix is zero in this homomorphic image to avoid redundant computation. In order to minimise data dependencies in the algorithm we do not already check the determinant when computing `noOfPrimes`. If some images cannot be used for constructing the result, the CRA will evaluate more results by demanding a so far unevaluated list element of `xList`.

### 6.3 Summary

In order to describe nested parallelism the higher-order nature of strategies is important. In the case of LinSolv an outer strategy defines the parallelism over `xList` with the strategy `par_sol_strat` as argument that defines the parallelism over the elements of this list. This demonstrates the use of data-oriented parallelism: the parallel behaviour is defined over the generated data structures rather than inside the functions that operate on the data structures. This approach, which exploits the non-strict semantics of data structures, allows the programmer to maintain a high degree of modularity when controlling the parallelism of the program. In the case of LinSolv it was sufficient to apply the strategy in Figure 7 to the top level function without changing the algorithm at all.

A direct comparison with a pre-strategy version of this program shows that the performance tuning of the parallel program is greatly facilitated by the separation between algorithmic and behavioural code. This is demonstrated by the development of three parallel versions of the code in [9]. In contrast to the strategic version, the pre-strategy version of the code combined the computation of the result with a specific dynamic behaviour suitable for parallelism, yielding rather opaque parallel code.

## 7 Discussion

This paper has described the construction of several medium-scale parallel functional programs, each program typifying a different application area. The construction has exposed several issues:

*Real Application Parallelism.* The accident blackspots application achieves good wall-clock speedup and acceptable scale-up on both a distributed-memory machine and a shared-memory machine. Both LinSolv and the Alpha-Beta search achieve good speedups on the simulator in realistic settings and their final versions demonstrated similar wall-clock speedups on a four processor shared-memory machine. The programs use a number of different parallelism paradigms, e.g. pipelining, divide-and-conquer and data-oriented parallelism, often combining several paradigms in a single program leading to irregular parallelism. From our experience it is too restrictive to support only certain paradigms without the possibility to combine and nest them. The regular use of higher-order strategies in this paper emphasises this aspect.

*Language Issues.* The clear separation of behavioural and algorithmic code was essential for the performance tuning of the algorithms. This is even more so in programs like Alpha-Beta search or LinSolv that make essential use of laziness. It was far easier to test about a dozen different variants of the strategic version of LinSolv than to change the code in various modules in order to achieve different parallel behaviour in the pre-strategy version.

The data-oriented parallelism used in the presented algorithms is in stark contrast to the control-oriented parallelisation typically used in imperative languages. The definition of parallelism over intermediate data structures increases the modularity of the parallel algorithm because the functions operating on the data structure can remain unchanged. A detailed discussion of data-oriented parallelism and a comparison between the strategic approach and imperative parallel programming can be found in [8]. As further work it would be interesting to investigate the use of strategies in a strict language or to use strategies as a parallel coordination language for components written in a strict language.

The definition of our strategies module evolved over the course of parallelising these programs. In particular, strategic function composition was added in order to facilitate the specification of data-oriented parallelism. This highlights the importance of using large applications for testing a programming technique like evaluation strategies.

*Engineering Environment.* Developing these programs has refined GranSim, GUM, and the profiling tools. For example, the design of the GranSim-Light setup providing idealised simulation was mainly motivated by our experiences in parallelising programs like LinSolv and the Accident Blackspots program. Considering that only a few systems for parallel functional programming have been engineered beyond prototype stage, it is particularly gratifying that the implementation technology is available on several architectures and handles programs as large as 47,000 lines. Our parallelisation methodology uses both sequential and parallel profilers, and the fact that GranSim is parameterisable is crucially important. The possibility to simulate an idealised machine as well as several different target architectures, and to run the parallel program in the same integrated environment greatly reduces the time needed to develop and tune a parallel program. It also allows the programmer to use the same visualisation tools for both simulation and parallel execution.

Although our visualisation tools have been essential in understanding the dynamic behaviour of these programs, the parallelisation process also demonstrated the need for better parallel profilers. In particular we would like to relate the active threads at any given time in a program execution to the strategy that created them. This observation was the main motivation for designing the parallel profiler GranCC, which is currently under development [2].

## References


1. 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, CO, Apr. 10–12, 1995.
2. K. Hammond, H-W. Loidl, and P.W. Trinder. Parallel Cost Centre Profiling. In *Glasgow Workshop on Functional Programming*, Ullapool, Scotland, Sep. 15–17, 1997.
3. 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, Dec. 1995.
4. R.J.M. Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2):98–107, Apr. 1989.
5. M. Lauer. *Computing by Homomorphic Images*, pp. 139–168. In Computer Algebra — Symbolic and Algebraic Computation. Springer-Verlag, 1982.
6. J. D. Lipson. Chinese Remainder and Interpolation Algorithms. In *SYMSAM*, pp. 372–391, 1971.
7. 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 — Intl. Workshop on the Implementation of Functional Languages*, Univ. of St. Andrews, Scotland, Sep. 10–12, 1997.
8. H-W. Loidl. *Granularity in Large-Scale Parallel Functional Programming*. PhD thesis, Dept. of Computing Science, Univ. of Glasgow, Mar. 1998.
9. H-W. Loidl. LinSolv: a Case Study in Strategic Parallelism. In *Glasgow Workshop on Functional Programming*, Ullapool, Scotland, Sep. 15–17, 1997.
10. 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, Jul. 1991.
11. A.P. Poulovassilis and C. Small. A Domain-Theoretic Approach to Logic and Functional Databases. In *VLDB'93 — Intl. Conf. on Very Large Databases*, pp. 415–426, 1993.
12. P. Roe. *Parallel Programming using Functional Languages*. PhD thesis, Dept. of Computing Science, Univ. of Glasgow, Feb. 1991.
13. 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, Jul. 5–7, 1993. Springer-Verlag.
14. 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, Mar. 1997.
15. P.W. Trinder, K. Hammond, H-W. Loidl, and S.L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1), Jan. 1998.
16. P. Trinder, K. Hammond, H-W. Loidl, S.L. Peyton Jones, and J. Wu. A Case Study of Data-intensive Programs in Parallel Haskell. In *Glasgow Workshop on Functional Programming*, Ullapool, Scotland, Jul. 8–10, 1996.
17. 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, May 1996.
18. J. Wu and L. Harbird. A Functional Database System for Road Accident Analysis. *Advances in Engineering Software*, 26(1):29–43, 1996.