Accidents always Come in Threes:
A Case Study of Data-intensive Programs in Parallel Haskell

P.W. Trinder, University of Glasgow
Glasgow, Scotland

K. Hammond, University of St Andrews
St. Andrews, Scotland

H-W. Loidl, S.L. Peyton Jones
University of Glasgow
Glasgow, Scotland

J. Wu
Centre for Transport Studies, University College London
London, England

Abstract

Accidents happen:

- "An invisible car came out of nowhere, struck my vehicle and vanished."
- "I pulled away from the side of the road, glanced at my mother-in-law, and headed for the embankment."
- "As I approached the intersection a sign suddenly appeared in a place where no stop sign had ever appeared before."

Luckily, we don't normally have to deal with problems as bizarre as these. One interesting application that does arise at the Centre for Transport Studies consists of matching police reports of several accidents so as to locate accident blackspots. The application provides an interesting, data-intensive, test-bed for the persistent functional language PFL. We report here on an approach aimed at improving the performance of this application using Glasgow Parallel Haskell.

The accident application is one of several large parallel Haskell programs under development at Glasgow. Our objective is to achieve wall-clock speedups over the best sequential implementations, and we report modest wall-clock speedups for a demonstration program. From experience with these and other programs the group is developing a methodology for parallelising large functional programs. We have also developed *strategies*, a mechanism to separately specify a function's algorithm and its dynamic behaviour.

1 Introduction

It has often been claimed that pure functional languages are highly suitable for parallel programming, but as yet there is little hard performance data to support such a contention, especially for non-trivial applications. The dearth of such programs is partly due to the lack of robust parallel implementations, and partly due to the absence of tools and techniques that support parallelisation.

We do now, however, have the publicly-available GUM runtime system for Haskell [21]. GUM stands for Graph-reduction for a Unified Machine-model, and has been ported to several parallel platforms, including the CM5 [6], Sun SPARCServer shared-memory multiprocessor and networks of Suns and Alphas. We also have the highly-tunable, and well-instrumented GranSim simulator that is based on the same compiler technology as GUM. This combination has allowed us to ensure reasonable and verifiable performance gains on a range of parallel platforms for relatively low implementation effort.

This paper describes work in progress that aims to explore the problems involved in writing non-trivial parallel programs, especially ones which are data-intensive – an area of special interest to our group. To date we have written a small demonstration program that achieves modest wall-clock speedups. We have also written a program to identify traffic-accident blackspots from real data, and are in the process of parallelising this program. We hope eventually to demonstrate real absolute performance gains for this program over its optimised sequential equivalent.

The traffic accident program is only one of several large Haskell programs that the group has written or parallelised, one of which (LOLITA, a natural language processing system [14]) approaches 100,000 lines of source. Our experiences with large-scale parallel functional programming have led us to develop a methodology for parallelising programs using simulation and both sequential and parallel profiling tools. We have also developed *strategies*, a new mechanism for specifying parallelism in non-strict functional programs. Properties of both parallelism and strictness can be expressed using the higher-order dynamic control provided by strategies. Our programs have also uncovered the need for new parallel profiling techniques.

The remainder of this paper is structured as follows. Section 2 briefly outlines Glasgow Parallel Haskell, strategies, and the methodology we have developed for writing parallel Haskell programs. Section 3 describes and gives the results obtained for the demonstrator program. Section 4 describes the traffic accident case study, giving our current results to date. Section 5 covers related work. Finally, Section 6 discusses our results.

2 Apparatus and Method

2.1 Glasgow Parallel Haskell

2.1.1 Language Constructs

The only language constructs used to introduce parallelism in Glasgow Parallel Haskell are the `par` and `seq` pseudo-functions. At present these combinators are added by the programmer, though we would of course like this task to be automated, and are actively pursuing research with this objective in mind [12]. The `par` combinator is a form of parallel composition, returning it's second argument. Operationally, when the expression `x `par` e` is evaluated, the heap object referred to by the variable `x` is *sparked*, and then `e` is evaluated. A common idiom (though by no means the only way of using `par`) is to write

```
let x = f a b in x `par` e
```

where `e` mentions `x`. Here, a *thunk* (or suspension) representing the call `f a b` is allocated by the `let` and then sparked by the `par`. It may thus be evaluated in parallel with `e`.

Rather than being an explicit control instruction, a spark is an indication that a thunk *might* usefully be evaluated in parallel, not that it *must* be evaluated in parallel: the implementation is free in principle to delay, ignore or even discard sparks depending on the dynamic characteristics of the system. For consistency, however, the results reported here do not discard sparks. Sparked thunks are similar to MultiLisp or Tera C *futures* [9], and the process of converting a spark to a thunk is similar to lazy task creation [13].

Sparking a thunk is a relatively cheap operation, consisting only of adding a pointer to the thunk to a processor's *spark pool*. The principal performance penalty arises, in fact, from the need to create the thunk in order to place it in the spark queue: in many cases, the sequential implementation could avoid this cost. Rushall has described a technique that avoids this problem for divide-and-conquer applications [19], but this does not seem to generalise to the irregular parallelism present in the transport-accident application, for example, and so we have not adopted this idea.

The `seq` combinator implements sequential composition. When the expression `e1 `seq` e2` is evaluated, `e1` is evaluated to weak head normal form first, and then the value of `e2` is returned. In the following parallel `nfib` function, `seq` is used to *force* the evaluation of `n2` before the addition takes place. This is because Haskell does not specify which operand is evaluated first, and if `n1` was evaluated before `n2`, there would be no parallelism.

```
nfib :: Int -> Int
nfib n | n <= 1    = 1
       | otherwise = n1 `par` (n2 `seq` n1 + n2 + 1)
                     where
                       n1 = nfib (n-1)
                       n2 = nfib (n-2)
```

Parentheses have been included for readability, but aren't actually necessary: ``seq`` binds more tightly than ``par``.

2.1.2 Implementation

Glasgow Parallel Haskell requires almost no changes to the Glasgow Haskell Compiler (ghc), instead it is supported by a portable parallel run-time system, GUM [21]. GUM is message-based, and portability is facilitated by using the PVM communications harness that is available on many multi-processors. As a result, GUM is available for both shared-memory (Sun SPARCserver multiprocessors) and distributed-memory (networks of workstations) architectures. The high message-latency of distributed machines is ameliorated by sending messages asynchronously, and by sending large packets of related data in each message.

Approximately 10 non-trivial programs have been run on GUM, though without being tuned to yield optimal parallel performance. Initial performance figures for simple programs demonstrate absolute speedups relative to the best sequential compiler technology. To improve the performance of a parallel Haskell program, GUM provides tools for monitoring and visualising the behaviour of threads and of processors during execution, based on those provided by the GranSim simulator.

2.1.3 GranSim

GranSim simulates the execution of parallel Haskell programs on a variety of architectures [8]. GHC's code generator is slightly modified to instrument the code with data locality and timing information. The simulation maintains a clock for each simulated processor and is event-driven, producing a log of events. Runtime-system options allow GranSim to simulate a wide range of different parallel architectures, different processors and optionally special features like thread migration. A suite of tools allows the visualisation of the parallel execution in several different ways: across the whole machine, by processor or by thread. Another suite of tools visualises the granularity of the generated threads. GranSim profiles of two programs are given in Sections 3 and 4.3.

GranSim and GUM were developed contemporaneously and build on mutual experience. Specifically, both systems use the same method for synchronising parallel threads and to a large extent the same code for communicating data. GranSim simulates some features not present in GUM, e.g. different methods for packing data for communication between processors and synchronous, incremental communication.

Parameterising GranSim to simulate different architectures is extremely valuable when parallelising large programs. Early stages of parallelisation use GranSim-Light: an idealised machine with zero-cost communication and an infinite number of processors. Later stages use GranSim parameterised to simulate a more realistic machine and address issues such as thread set-up costs and communication delays. The use of GranSim is discussed in more detail in Section 2.3.

2.2 Strategies

In Glasgow Parallel Haskell a function must both describe the value to be computed, and its *dynamic behaviour*. The dynamic behaviour of a function has two aspects: *parallelism*, i.e. what values could be computed in parallel, and *evaluation-degree*, i.e. how much of each value should be constructed. Our experience is that the semantics of a function can be obscured if its dynamic behaviour is intertwined with its functional definition. Strategies have been developed to address this problem: a strategy is an *explicit* description of a function's potential dynamic behaviour, separated from its functional specification. Our philosophy is that *it should be possible to understand the semantics of a function without considering its dynamic behaviour*.

In essence a strategy is a runtime function that identifies the subcomponents of an expression that need to be evaluated, describes the degree of evaluation for each subcomponent and possibly sparks a thread to perform the evaluation in parallel. Because strategies are functions they can be defined on any type and can be combined to specify sophisticated dynamic behaviour. For example, a strategy can control thread granularity or specify evaluation over an infinite structure. A disadvantage is that a strategy requires an additional runtime pass to be made over some data structures.

2.2.1 Definition

Because a strategy specifies only the dynamic behaviour of a function it has no semantic value, it does however return a null value to indicate completion, i.e.

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

A strategy is typically applied by the `using` function below.

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

The `using` function allows the strategy to be specified as 'just another' local definition, i.e. separated from the specification of the algorithm. For example we can write `pfib` as follows.

```
pfib n
  | n <= 1    = 1
  | otherwise = n1+n2+1 `using` strategy
                where
                  n1 = pfib (n-1)
                  n2 = pfib (n-2)
                  strategy r = n1 `par` n2 `seq` ()
```

Furthermore, `` x `using` s `` is a *projection* on `x`, i.e. both

- *a retraction* – `` x `using` s `` is more defined than `x`, or `` x `using` s `` $\sqsupseteq$ `x`.
- and *idempotent* – `` (x `using` s) `using` s = x `using` s ``

2.2.2 Basic Strategies

The simplest strategies introduce no parallelism: they specify only the degree of evaluation. The simplest evaluation-degree is termed `r0` and performs no reduction at all. This is surprisingly useful, e.g. when evaluating a pair the first element can be evaluated but not the second – as is done in LOLITA [14]. `r0` is easily defined for all types:

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

A value can be reduced to Weak Head Normal Form (WHNF), by `rwhnf`. Like most non-strict functional languages Haskell uses WHNF reduction by default, making it easy to define `rwhnf` for all types:

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

A *data value* (but not a function value) can also be reduced further to *normal form* (NF) using `rnf`. The distinction between data and functional values is captured by defining `rnf` as a method of class `NFData`, which is analogous to the `Eval` class in Haskell 1.3. Because NF and WHNF coincide for base types like integers and booleans, the default method for `rnf` is `rwhnf`.

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

For constructed types an instance must be declared specifying how to reduce a value of that type to normal form. Such an instance relies on its element type 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
```

So it is possible to define `rnf` for lists of pairs of integers, but not for lists of functions etc.

In a parallel `quicksort`, each parallel thread should construct all of its result list, and `rnf` expresses this neatly.

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

2.2.3 Combining Strategies

Because strategies are simply functions, they can be combined using the full power of the language, e.g. passed as parameters or composed using the function composition operator. For example, `seqList` 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. There is an analogous function for every constructed type.

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

A strategy can specify parallelism, and typically also specifies the degree of evaluation. 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)
```

2.2.4 Using Strategies

Strategies allow program behaviour to be defined independently of the values that are defined in the program. In the definition of `parMap` below, the algorithm has been specified without considering its dynamic behaviour, as `map f xs`. The parallelism we wish to introduce is captured by `parList`, and the strategy to be applied to each element of the result list is specified by the `strat` parameter.

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

2.3 A Methodology for Parallelising Programs

The recipe that is emerging for parallelising large non-strict functional programs is sketched below. 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.

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

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

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

4. **Parallelise Big Eaters** using 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. **Simulate First.** Using an idealised simulator like `hbcpp` or GranSim eliminates some of the complexities of a real parallel implementation, like task migration, communication times etc. This is a "proving" step: if the program isn't parallel on an idealised machine it won't be on a real machine. A simulator is often easier to use, more heavily instrumented and can be run on a workstation.

6. **Simulate Second.** 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 set-up costs.

7. **Real Machine.** 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 historically often proved highly frustrating: the development environments on parallel machines are often 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 artefacts of the communication system or other dynamic characteristics.

3 Demonstrator

Unless we can achieve wall-clock speedups for a simple data-intensive program we have failed. The demonstrator also allows us to gain experience of parallelising Haskell programs, uncover bugs in the GUM runtime system.

3.1 Bill of Material Explosion

Date poses the following bill of material, or parts explosion, problem [5]. Given the relation below, write a program to list all the component parts of a given part to all levels.

PARTS

Main Component	Sub-Component	Quantity
P1	P2	2
P1	P4	4
P5	P3	1
P3	P6	3
P6	P1	9
P5	P6	8
P2	P4	3

A naive function `explode` lists the components of a single part, `main`. The program generates a bill of material (a list of tuples), then explodes a sequence of part numbers before printing the number of parts in each explosion.

```
explode parts main = [p | (m,s,q) <- parts, m == main, p <-
(s:explode parts s)]

doQuery lo hi bomSize = map length explodeList
                        where
                          bom = generate bomSize
                          explodeList = map (explode bom) [lo..hi]
```

The program is inherently data parallel because the explosion of one part is not dependent on the explosion of any other part. Constructing the bill of material in memory is atypical of a query program: a more realistic program would read it in from a file. For this reason we do not parallelise the construction `generate`. Once the bill exists, the parts are exploded in parallel. This dynamic behaviour is specified by adding a strategy to `doQuery`:

```
doQuery lo hi bomSize = map length explodeList `using` strategy
                        where
                          bom = generate bomSize
                          explodeList = map (explode bom) [lo..hi]
                          strategy result = (rnf bom) `seq`
                                            (parList rnf explodeList)
```

The program's semantics and dynamic behaviour have been chosen carefully. The parallelism is *inter*-explosion, rather than *intra*-explosion, because the granularity, i.e. thread duration, is too small if individual explosions are parallelised.

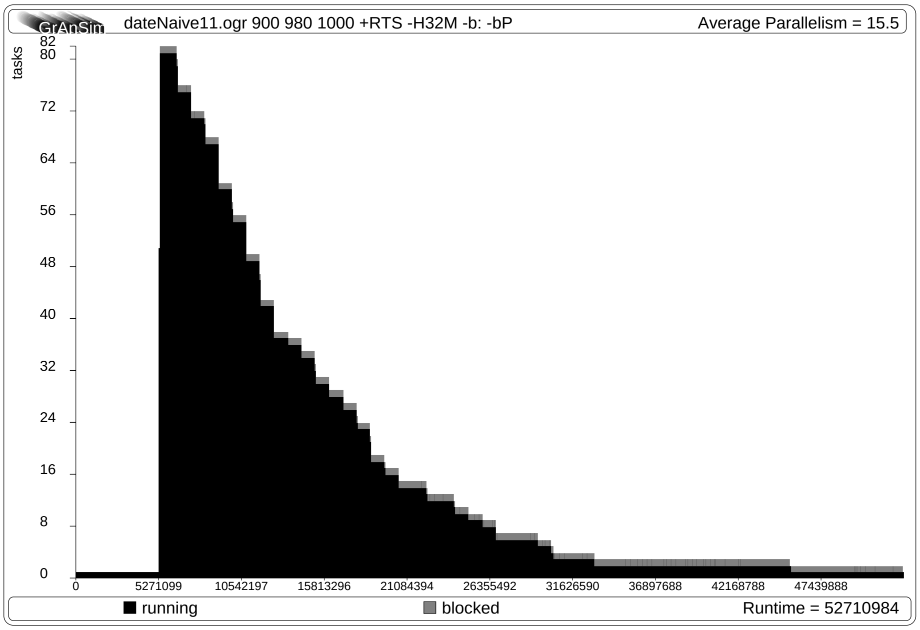

Figure 1: Bill of Material GranSim Activity Profile

To focus attention on the interesting part of the computation, only the number of parts in an explosion is printed: even though all of the explosion is constructed by `parList rnf explodeList`. If all parts were printed, a long sequential tail would be observed at the end of the computation. This occurs because I/O to the console is both slow and sequential.

3.2 Results

In the examples presented here the bill of material contains 2000 elements, each part has 3 sub-components, some of which may in turn have subcomponents. 80 parts are exploded, and the sizes of the first 10 explosions output are 40, 45, 45, 30, 12, 3, 14, 3, 12 and 17. The runtime of the optimised sequential code on a Sun SPARC 10 is 25.1 seconds.

Because of its simplicity, the bill of materials program does not need to use all of the steps in the methodology outlined in Section 2.3. As the bill construction has not been parallelised, there is no top-level pipeline. The computational costs of the program are also simple enough to understand without time profiling.

We simulate first, and Figure 1 shows the GranSim activity profile for an idealised machine. The initial sequential segment is the bill construction, representing less than 1% of the work. When the parts are exploded in parallel, maximum parallelism is rapidly attained as there are no dependencies between the explosions. Some explosions contain more parts, and so take longer to complete, and this leads to the raggedly-declining parallelism that we observe. The two threads running between 32×10^6 and 44×10^6 cycles are firstly a printing thread and secondly a part explosion that contains 116 components. The final sequential tail is due to printing the output. These conclusions can be verified using a per-thread visualisation of the execution.

We simulate second, and Figure 2 shows the activity profile when GranSim is parameterised to resemble the target machine. The average parallelism of 3.8 indicates that the program is adequately parallelised for a 4-processor machine.

The parallel machine used to execute this program is a Sun SPARCserver with six SPARC 10 processors, communicating via shared memory segments under Solaris 2. The machine is heavily loaded, seldom having less than 30 users, or a load average below 3. Figure 3 shows the wall-clock speedups for the bill of material program. GUM does not support thread migration, and hence execution is highly susceptible to sheduling accidents [21]. To ameliorate the effects of such accidents, each point on the graph is the median of three separate execution times. The parallel code has

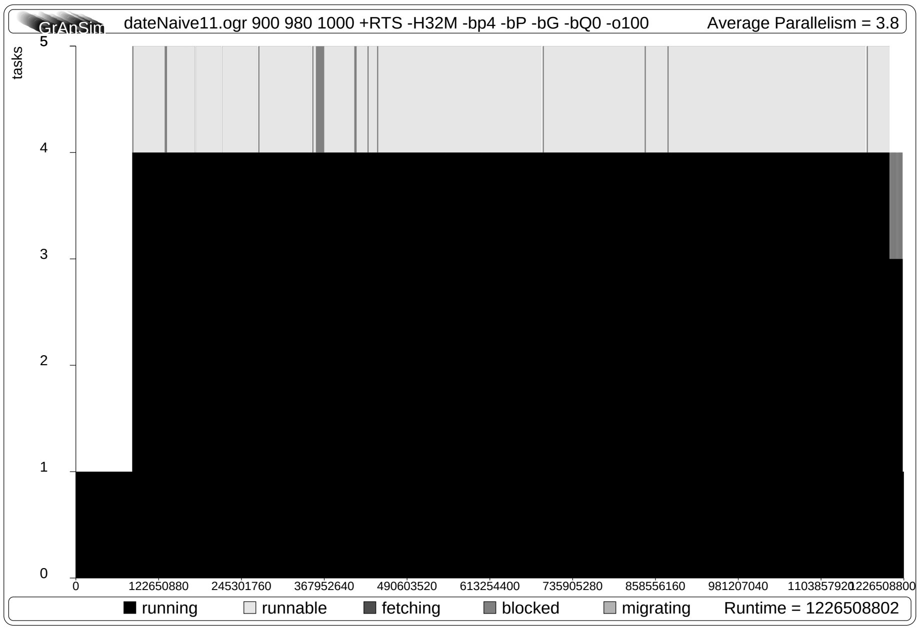

Figure 2: Bill of Material GranSim Activity Profile

some overheads that are not present in the sequential Glasgow Haskell implementation, such as the need to test every closure on entry to determine if it is already being evaluated by another task. As a result, the parallel code on a single processor slows down by 20% to make the parallel system only 80.5% as efficient as the sequential implementation. A careful analysis of the efficiency of GUM can be found in [21].

Because of the load on this machine, it is not possible to use all six processors for any significant length of time. Hence the first plot shows wall-clock speedups for just four processors on a lightly loaded machine. The second plot shows the speedups achieved when the program is run on up to six processors. In the second case, the machine is more heavily loaded (approx 30 users).

A wallclock speedup of 2.2, and relative speedup of 2.6, on a busy four-processor machine is encouraging for a program with a significant sequential tail. It is clear, however, that the machine used for the performance tests, is too heavily loaded to give a realistic indication of the potential parallel performance that could be achieved. We would expect results on an unloaded machine to be significantly better than those reported here.

4 Traffic Accident Case Study

4.1 Description

Given a set of independently produced police accident records, the task is to discover any accident blackspots (those locations where a number of accidents 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

1. at the same junction number,
2. at the same pair of roads,
3. at the same grid reference, or
4. within a small radius of each other. The radius is determined by the class of the roads, type of the junction etc.

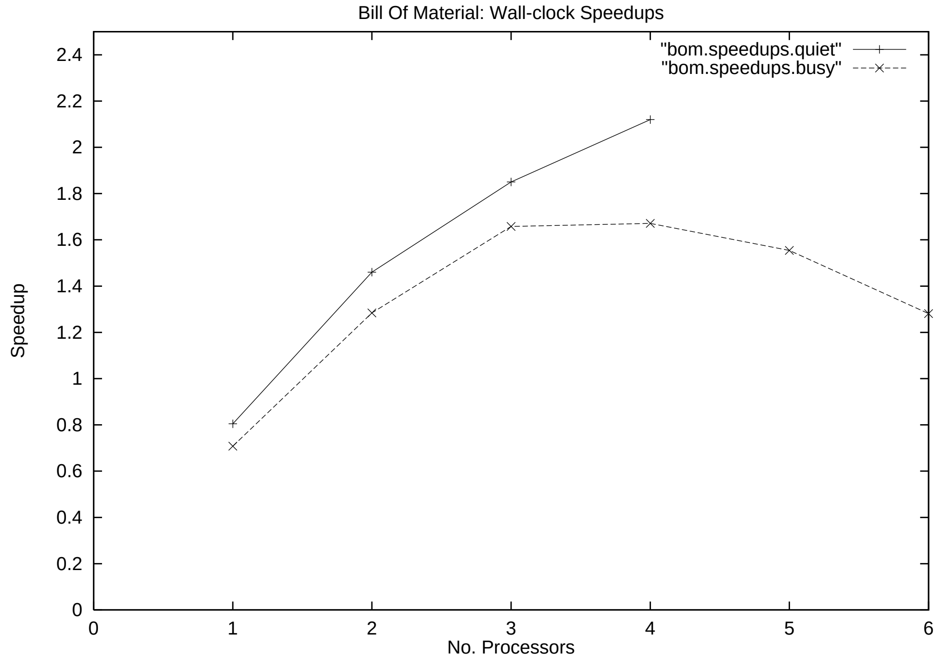

Figure 3: Bill of Material: Wallclock Speedups

The problem amounts to partitioning a set into equivalence classes. Partitioning on a single condition `f` is easy: `{{a' |a' <- acc /\ f a a'} | a <- acc}`. However, combining a number of partitions introduces transitivity. 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) [20]. 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.

4.2 PFL and Haskell Implementations

The application was originally written by Jiashu Wu [23] in PFL and has subsequently been rewritten in Haskell. PFL is an interpreted functional language [17], designed specifically to handle large deductive databases. Unusually for a functional language, PFL provides a uniform persistent framework for both data and program. Bulk data are accessed and updated through a special class of updatable functions called selectors, and functions can be defined incrementally in multiple user sessions by the insertion and deletion of equations which are stored in the database.

By exploiting persistence, PFL supports applications involving arbitrarily large volumes of both data and programs (subject only to the availability of secondary memory and the degradation in performance). From a deductive database perspective, selectors represent extensional relations, whilst further functions defined over selectors act as derivation rules representing intensional relations. The expressiveness of PFL and the uniform functional formalism for both data storage and manipulation considerably simplifies the implementation of complex queries.

4.2.1 The PFL Implementation

Wu's PFL implementation of the traffic accident application comprises approximately 500 lines of PFL. The partition is achieved in the following stages.

1. Generate a partition for each of the four accident location criteria outlined above. For the first three criteria, each partition$_i$ can be obtained through a straightforward database retrieval with an appropriate pattern match:

   ```
   partition_i = {{a | a <- acc, f a = v}
                 | v <- attributeValues_i} (i=1,2,3)
   ```

   This can be read as a nested set comprehension. For the final criterion, a function that searches nearby accidents for a given accident must be applied to each accident:

   ```
   partition_4 = {{searchNearByAcc a} | a <- acc}
   ```

2. Convert each partition into a binary "sameSite" relation by pairing the least accident in a subclass with every other. The least accident is termed the accident index. We label the subsets of `partition_i` `p_ij`, for 1 ≤ `j` ≤ `n_i`:

   ```
   sameSite_i =   U      {(l,a) | a <- p_ij, a != l}
               j = 1..n_i    where
       l = least p_ij
   ```

3. Merge the four `sameSite` relations into a single one using a set union operation:

   ```
   sameSite = sameSite_1 U sameSite_2 U sameSite_3 U sameSite_4
   ```

4. Finally, the combined partition is formed by repeatedly finding all of the accidents reachable in `sameSite` from each given accident index. This is done destructively: R' represents R with all pairs containing a removed.

   ```
   partition     = {reachable a sameSite | a <- indexAcc}
   reachable a R = {m | a R x \/ x R a, m <- {x} U reachable x R'}
   ```

All relations involved in the above four steps are represented by selector functions so as to take advantages of PFL's concise notations for querying bulk data values.

4.2.2 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:

```
sameSite = {(a,a')|a <- acc, a' <- acc, f1 a a' \/ ... fn a a'}
```

The combined partition is formed by repeatedly finding all of the accidents reachable in `sameSite` from a given accident.

```
partition     = {reachable a sameSite | a <- acc}
reachable a R = {m| a R x \/ x R a, m <- {x} U reachable x R}
```

The Haskell implementation uses data that has been transformed to protect privacy, but preserves the essential relationships. The program has three major phases: reading and parsing the file of accidents; constructing sameSite, and indices over accident and sameSite; forming the partition. The sequential implementation is a 286-line module, together with 3 library modules totalling 1300 lines. The parallel implementation is 515 lines in 2 modules together with 4 library modules totalling 1500 lines.

The results and runtimes of the PFL and Haskell programs are given in Table 4.2.2. The runtimes are only for broad comparison purposes as the programs were run on similar, but not identical machines: PFL on a Sun ELC, and Haskell on a Sun Classic. There are 7310 accidents, runtimes are given in minutes and seconds and we abbreviate multiple accident sites as MA sites.

SEQUENTIAL RESULTS

Partition by	MA Sites	Accidents at MA Sites	PFL Run Time, m:s	Haskell Run Time, m:s
Junction No. only	208	1617	2:41	0:47
Road-pair only	1310	4653	14:41	0:52
Grid ref. only	1324	4434	14:16	0:52
Junction No, Road-pair and Grid ref	1228	5430	17:44	1:42
Junction No, Road-pair, Grid ref. and nearby	1229	5450	18:25	2:03

We expect Haskell compiled with the Glasgow compiler (ghc) to be one or two orders of magnitude faster for CPU-intensive problems than the interpreted PFL. However, in contrast to PFL's carefully designed persistence, ghc's lazy file I/O is poor: amongst other things, requiring the data to be parsed from text on input. The Haskell run times for the single-condition partitions are very similar. Most of the time is probably absorbed in reading and parsing the file of accidents. For programs with more computation over the data, like the more complex partitions, Haskell's greater execution speed becomes apparent.

To avoid creating an unfair impression from this set of timings, it is important for the reader to note that this is not a typical PFL application, being essentially memory-resident once the initial data is read. PFL's primary application domain concerns problems where the data is normally too large to fit into primary memory. We would not expect the present Haskell implementation to cope well (if at all) with such problems!

4.3 Parallelisation

The Haskell implementation is being parallelised, having reached the simulate-first stage of the methodology. The initial target machine is the SPARCserver, with only 4 usable processors, so the immediate objective is a uniform average parallelism of around 6. The average parallelism obtained at several stages is as follows:

- **1.2 tasks: Top Level Pipeline** is disappointingly low. The problem is that the indices are trees, and all of the tree must be constructed before it can be consumed by the next pipeline stage.
- **1.4 tasks: Parallel Partition** The partition is parallelised using benign speculation: the equivalence classes of n accidents are computed in parallel. If two or more of the accidents are in the same class, the work is duplicated. The chance of wasting work is small as the average class size is 4.4, and there are over 7000 accidents. The speculation is benign because the amount of work is bounded, and no other threads are sparked.
- **1.8 tasks: Improved parser and parallel file reading** After parallelising the partition the lazy file-reading and parsing were taking over 50% of the time. A parser specific to the accident file was constructed, and the data split into 4 files to be read in parallel.
- **2.2 tasks: Parallel Index Construction**
- **3.2 tasks: Improved Partition**
- **3.6 tasks: Split Indices** into a list of indexes that can each be accessed independently.

Figure 4 shows the GranSim activity profile for the accident program. Interpreting the activity profile of a large parallel program, such as this, can be quite subtle. It is easy to identify some parts of the program: the initial four threads are the file read, with index construction occurring on the results read so far. The band of 20-thread peaks are generated by the partition which speculatively finds 20 equivalence classes in parallel. However, it is not clear what part of the program is being evaluated by a particular thread at a particular time.

In GranSim it is possible to label threads created by a `par`. However, most `par`s are embedded within library strategies and a specific use of the strategy cannot be distinguished. Ideally we would like to identify which strategy is controlling the dynamic behaviour of the program at a given point in time. We have a rudimentary strategy labelling combinator, and are considering how to relate parallel cost-centre profiling to labelled strategies.

On an idealised machine with 4 processors, the file reading and partitioning are satisfactorily parallelised. The index construction must still be parallelised. The parallelisation is still at the simulate-first stage of the methodology.

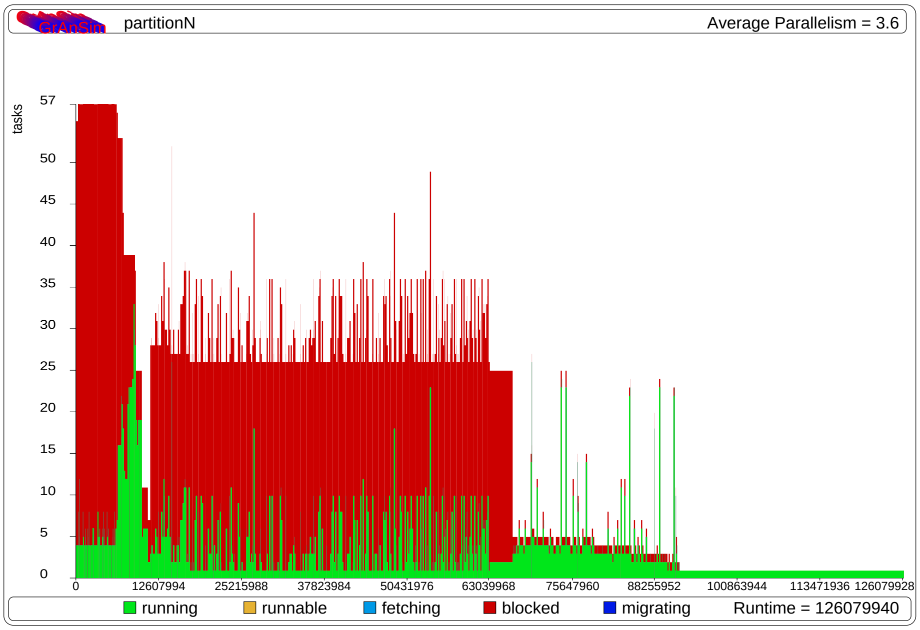

Figure 4: Accident GranSim Activity Profile

5 Related Work

The FLARE project [18] studied a number of realistic functional applications, yielding results for both simulations and for the GRIP multiprocessor. Programs included a computational fluid dynamics solver, a theorem prover, a telephone network routing and a graphical user interface. The results obtained for these programs showed that it was possible for non-expert programmers to develop serious parallel functional programs, capable of yielding some performance gain.

Experience gained with the FLARE applications and others (including a linear equation solving package [11] and a very large natural language system [14]) has paved the way for the methodology which we have expounded in this paper, particularly the use of the initial simulation phase.

The FLARE project used a Haskell-based idealised simulator *hbcpp* [22]. In its incarnation as GranSim-Light, GranSim yields results which are comparable to those provided by *hbcpp*, though it is much more accurate in costing both raw reduction and system overheads (this is important, since, as we have shown, I/O often causes serialisation). GranSim is, of course, also capable of providing several levels of more detailed simulation, as necessary for the development of the parallel application. There is, naturally, some cost associated with obtaining this extra information: but this cost is proportional to the information gained.

Like GUM, DIGRESS [3] also provides a parallel implementation of Haskell running on distributed workstations, with dynamic scheduling and global load-balancing. DIGRESS is, however, designed specifically for this environment, and so provides features such as automatic system reconfiguration as new workstation resources become available. We are not aware of comparative performance figures for the two systems, however, DIGRESS appears to be designed primarily as a platform for experimentation with parallelism rather than for ultimate performance.

pH is another well-known parallel Haskell, developed at MIT [15]. It differs from Glasgow Parallel Haskell in being entirely implicit, and in modifying the Haskell language to support dataflow ideas such as I-structures and k-bounding for loops. The pHluid implementation [7] uses dataflow techniques developed for Id (essentially, this is a pH front-end with an Id back-end), so performance is not yet as good as for Glasgow Haskell.

Several authors have suggested higher-order programming techniques to control parallelism. The best-known of these techniques are probably based on algorithmic skeletons [4]. Skeletons differ from strategies in that they tightly integrate control and value. In effect, good patterns of parallel behaviour are associated with the use of certain higher-order functions. The corresponding behaviours are invoked automatically at runtime when these functions are called (normally based on static compilation techniques). The approach is more implicit than that suggested here, in that behaviours are chosen based only on the functions that are used. Programmers must, however, cast their programs to suit the skeletons they intend to use, and it is not usually possible for the system to react to dynamic changes by choosing a different behaviour.

Evaluation transformers, as suggested by Burn [2] are also related to strategies in that they specify evaluation degree on a data structure, and can be used to control parallel behaviour. Unlike strategies, evaluation transformers specify a fixed degree of evaluation, implicitly invoked on the basis of strictness information to reduce values to the extent required. Evaluation transformers are intended to be automatically generated by the compiler rather than programmable, as here.

Like strategies, Caliban's `moreover` clause [10] also allows control to be separated from value under programmer control. Unlike strategies, moreover-clauses are statically evaluated with a view to creating a static map of tasks to processors.

6 Discussion

It has been well worth the effort to address a real problem. The work is still in progress, but we have already learnt much from it, and from the other large parallel programs being developed at Glasgow. The experience can be classified as functional data-intensive and parallel programming.

6.1 Experiences

Functional programming. Lazy time profiling helps identify the computation-intensive parts of the program worth parallelising. Lazy file reading and parsing data from text is excruciatingly slow. System libraries greatly reduce the implementation effort. Unfortunately, in order to define strategies over library types it is necessary to take private copies of the library modules.

Data-intensive programming. Coping with null values in a language without explicit support requires care. Multiple bulk types, in particular sets, lists and trees are useful, and both comprehesion notation and uniform function names would be a great help. There are also no built-in consistency checks or the concise and efficient access to bulk types as provided by PFL selectors. Compared to PFL, Haskell execution is fast, except, of course, for the crucial aspect of I/O operations.

Parallel programming. A methodology for parallelising large programs is emerging. It makes use of a parameterised simulator with heavy instrumentation and many ways of visualising the execution. We have also developed strategies, a mechanism for separately specifying a program's algorithm and its dynamic behaviour. Benign speculation again proves to be a useful means of parallelising functional programs.

6.2 Future Work

Work is continuing on the traffic accident program. Once good simulated speedups have been obtained, we intend to run the program on the Sun SPARCserver. To get good parallelism results however, we plan to port GUM to a less heavily-loaded machine. Strategies are being used at Glasgow, St Andrews, and Durham to parallelise several large programs. A paper describing strategies, and our experiences using them is nearing completion. In the longer term we would like to develop a parallel cost-centre profiling tool that relates a program's dynamic behaviour to its strategies.

References

[1] Blelloch G.E. and Greiner J. "A Provable Time and Space Efficient Implementation of NESL", *Proc. ICFP '96*, Philadelphia, Pennsylvania, (1996), pp. 213–225.

[2] Burn G.L.. "Evaluation Transformers — A Model for the Parallel Evaluation of Functional Languages (Extended Abstract)" *Proc. FPCA '87*, Springer-Verlag LNCS 274, (1987), pp. 446–470.

[3] Clack C. "Painless Parallel Programming.", *Parallel Processing in Engineering Community Club*, (1995).

[4] Cole M.I. *Algorithmic Skeletons: Structured Management of Parallel Computation*, Research Monographs in Parallel and Distributed Computing, Pitman, (1989).

[5] Date C.J. *An Introduction to Database Systems*, 4th Edition, Addison Wesley, (1976).

[6] Davis K. "MPP Parallel Haskell" *Proc. IFL '96*, Bonn, Germany, (September 1996) Springer Verlag (in press).

[7] Flanagan C. and Nikhil R.S. "pHluid: the Design of a Parallel Functional Language Implementation", *Proc. ICFP '96*, Philadelphia, Pennsylvania, (May 1996), pp. 169–179.

[8] Hammond K., Loidl H-W., and Partridge A.S. "Visualising Granularity in Parallel Programs: A Graphical Winnowing System for Haskell", *Proc. HPFC'95 — High Performance Functional Computing*, Denver, Colorado, (April 1995), pp. 208–221.

[9] Halstead R.H. "Multilisp - a language for concurrent symbolic computation" TOPLAS 7(4) (October 1985), pp 501-538.

[10] Kelly P.H.J. *Functional Programming for Loosely-coupled Multiprocessors*, Research Monographs in Parallel and Distributed Computing. Pitman, (1989).

[11] Loidl H.-W. and Hammond K. "Solving Systems of Linear Equations Functionally: a Case Study in Parallelisation", Technical Report, University of Glasgow, (1994).

[12] Loidl H.-W. and Hammond K., "A Sized Time Analysis for a Parallel Functional Language", *This Proceedings*, 1996.

[13] Mohr E., Kranz D.A., Halstead R.H., "Lazy Task Creation – a Technique for Increasing the Granularity of Parallel Programs", *IEEE Transactions on Parallel and Distributed Systems* **2(3)** (July 1991), pp. 264–280.

[14] Morgan R.G. and Jarvis S.A. "Profiling Large-Scale Lazy Functional Programs." *Proc. HPFC'95 — High Performance Functional Computing*, Denver, Colorado, (April 1995), pp. 222-234.

[15] Nikhil R.S., Arvind, Hicks J., Aditya S., Augustsson L., Maessen J.-W. and Zhou Y. "pH Language Reference Manual, Version 1.0 – Preliminary." Computation Structures Group Memo 369, Laboratory for Computer Science, MIT, (January 1995).

[16] Poulovassilis A.P., and Small C. "A Functional Programming Approach to Deductive Databases", *Proc. 17th Intl. Conf. on Very Large Databases (VLDB '91)*, G. Lohman, et al. (eds), (1991), pp. 491–500.

[17] Poulovassilis A.P., and Small C. "A Domain-Theoretic Approach to Logic and Functional Databases", *Proc. 19th Intl. Conf. on Very Large Databases (VLDB '93)*, (1993), pp. 415–426.

[18] Runciman C. and Wakeling D. (eds.), *Functional Languages Applied to Realistic Examplars: the FLARE Project*, UCL Press, (1994).

[19] Rushall D. *Task Exposure in the Parallel Implementation of Functional Programming Languages*, PhD Thesis, Dept. of Comp. Sci., University of Manchester, (1995).

[20] Tarjan R.E. "Efficiency of a good, but not linear set union algorithm" J. ACM 22 (1975), pp. 215-225.

[21] Trinder P., Hammond K., Mattson J., Partridge A., and Peyton Jones S.L. "GUM: a Portable Parallel implementation of Haskell". Proceedings of Programming Languages Design and Implementation, Philadelphia, USA, (May 1996).

[22] Wakeling D. and Runciman C., "Profiling Parallel Functional Computations (Without Parallel Machines)". *Proc. 1993 Glasgow Workshop on Functional Programming*, Springer-Verlag WICS, (1993), pp. 235–248.

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