# Towards a Parallel Search for Solutions of Non-deterministic Computations

Fabian Reck and Sebastian Fischer

Department of Computer Science
Christian-Albrechts-University of Kiel
D-24118 Kiel
{fre,sebf}@informatik.uni-kiel.de

**Abstract:** In this paper we explore the possibilities to perform the search for results of non-deterministic computations in parallel. We present three different approaches to this problem: Dividing the search space statically, using a Bag of Tasks approach and using semi-explicit parallelism. Then we discuss their advantages and limitations. Finally, we discuss benchmarks using a monadic SAT solver as an example of a non-trivial application of a non-deterministic program.

## 1 Introduction

In recent years, computers are mainly made faster by adding more and more cores. On the other hand, many programs written nowadays make use of only one core. Probably one of the reasons for this situation is that writing concurrent or parallel applications is difficult and error prone. An approach to improve this situation is to provide primitives that allow the programmer to add parallelism to his programs without reasoning about threads, locking, etc.

In this paper we discuss an approach to parallelize the execution of functional logic programs. Together with a new way to write purely functional lazy logic programs in Haskell [FKS09], and our recent plans to translate programs written in the functional logic language Curry [Cur] to Haskell exploiting the results of Fischer, Kiselyov and Shan [FKS09], we expect that our approach will help to speed up the execution of functional logic programs.

There are two ways to parallelize such programs. The first is to parallelize computations that are needed for a single result, e.g., both arguments of an equality test. The second is to compute different results of a non-deterministic computation in parallel. In this paper we want to approach the second way. We discuss three different algorithms to enumerate the results of non-deterministic computations in parallel.

In Section 2 we give a brief introduction to non-determinism in the purely functional language Haskell. Then, in Section 3, we develop a tree representation of non-deterministic computations and present three approaches to parallel search using this representation: di-

viding the tree between several threads (3.1), using a bag of tasks approach (3.2) and using semi-explicit parallelism (3.3). A simple implementation of a monadic SAT solver serves us as an example for a non-deterministic application in section 4. In Section 5 we use the same SAT solver to benchmark our three search algorithms and discuss the results. We mention related work in Section 6 and give some ideas about future work in Section 7. Finally, in Section 8, we conclude.

## 2 Non-determinism

In logic languages such as Prolog, and functional logic languages such as Curry [Cur], non-determinism is a key feature. Non-determinism means that a single expression can be evaluated to more than one value.

But non-determinism is also known in purely functional languages such as Haskell [Has]. In functional languages expressions are not allowed to evaluate to more than one value, but an expression can evaluate to a data structure that represents multiple non-deterministic values. In Haskell the best known example for such a data structure is the list. Every function

```
f :: T1 -> [T2]
```

can have two meanings: The first and obvious is that it takes a value of type `T1` and produces a list of values of type `T2`. The other, we are interested in here, is that a expression `f t` evaluates non-deterministically to a value of type `T2` and all such values are held in a list.

Philip Wadler proposed monads to structure programs [Wad92]. One of the monadic effects he discussed is non-determinism. The monad instance for lists is defined as follows:

```
instance Monad [a] where
  return a = [a]
  []     >>= _ = []
  (x:xs) >>= f = f x ++ (xs >>= f)
```

The `return` combinator lifts a value into the monadic context. Here it creates a singleton list holding this value. More interesting is the bind operator:

```
(>>=) :: m a -> (a -> m b) -> m b
```

It combines two monadic computations. It takes a monadic computation of type `a` and a function that takes a value of type `a` and returns a monadic computation of type `b`. The result is a monadic computation of type `b`. In our context the first argument is a non-deterministic computation and the second a function that takes *one* result of this computation and itself yields a non-deterministic result, i.e., a list. To get all the possible values of such a combined computation, the second argument of `(>>=)` has to be applied to every

result of the first computation. Then all of the results have to be concatenated. And that's exactly what the implementation shown above does.

With monads we have a generic way to create non-deterministic computations with a single result and to combine two non-deterministic computations. However, if we want to define a non-deterministic choice between two computations, we are stuck to the actual list operations, e.g., the concatenation of lists `return v1 ++ return v2`.

But there is a type class that provides the functionality we need here:

```
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

Interpreted as operations for non-determinism monads `mzero` represents a computation with no result, e.g., for lists it is the empty list. `mplus` is the non-deterministic choice between two computations, as seen in the example above, it is implemented as `(++)` for lists.

With the help of these two type classes it is possible to write non-deterministic code without tieing oneself to a specific implementation. For example, the following program non-deterministically computes a permutation of a given list:

```
perm :: MonadPlus m => [a] -> m [a]
perm []     = []
perm (x:xs) = perm xs >>= insert x

insert :: MonadPlus m => a -> [a] -> m [a]
insert x []     = return [x]
insert x (y:ys) = return (x:y:ys)
                  `mplus`
                  insert x ys >>= return . (y:)
```

The operation `perm` takes a list as argument and inserts the first element non-deterministically in the permuted rest list.

The operation `insert` takes two arguments: an element and the list into which it ought to be inserted. The result of `insert` is a non-deterministic choice, indicated by the operation `mplus` which is written infix, between the monadic action adding the element to the head of the list and the non-deterministic insertion of the element into the rest of the list.

The application of the operation `perm` to the list `[1,2,3]` has the type `MonadPlus m => m [Int]`. Which non-determinism monad `m` is used is determined by the context. For example, the expression `length (perm [1,2,3])` will yield the value `6` since length computes the length of lists and therefore the `MonadPlus` instance of list is used.

In the next section we present an instance of `MonadPlus` that is suitable for parallel processing. Hence, non-deterministic programs don't need to be written specially in order to be parallelized. It is sufficient to use the operations of the type class `MonadPlus`.

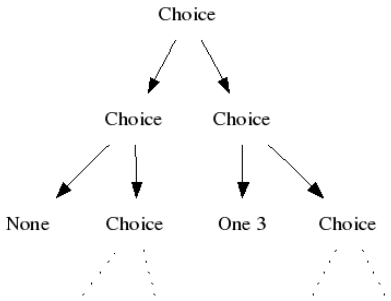


Figure 1: Tree structure of a non-deterministic computation

## 3 Approaches to a Parallel Search

Now we want to think about how to obtain all results of a non-deterministic computation `m a`. When we opt for the list as non-determinism monad, it seems as if we are already done. But note that Haskell is a lazy language and `m a` might be an expensive computation that is only evaluated when at least the structure of the list is demanded by the program. That can take quite some time, even if the resulting list is empty.

Since the results of a non-deterministic computation are independent, it should be possible to parallelize this search. At first we observe that a list is not an appropriate representation. As soon as we try to divide a list to evaluate it in parallel, the structure of the list is demanded and, therefore, the list is evaluated before we can distribute it for evaluation.

So we need another representation for non-deterministic data. A non-deterministic computation is either a failure (has no result), a single value or a choice between two non-deterministic computations. So the structure is that of a binary tree and can directly be represented as such[1]:

```
data SearchTree a = None | One a
                  | Choice (SearchTree a) (SearchTree a)
```

Figure 1 shows the graph representation of this structure. Now we need to impose a monadic structure on our new representation:

```
instance Monad SearchTree  where
  return = One

  None        >>= _ = None
  One x       >>= f = f x
```

[1]This implementation is available from Hackage as package tree-monad [tre]

```
Choice s t >>= f = Choice (s >>= f) (t >>= f)
```

The implementation of `return` is self explanatory, and `(>>=)` just replaces all occurrences of leaves `One x` with the tree resulting from `f x`. To make `SearchTree` an instance of `MonadPlus`, `mzero` is directly implemented as `None` and `mplus` as `Choice`.

Now we have a data structure that is more easily distributable for parallel evaluation. Note that laziness is very important in our approach. It allows the tree structure to be generated on demand during search and, hence, to parallelize its computation.

### 3.1 First Approach: Dividing the Tree Between Multiple Threads

Our first approach is the use of Concurrent Haskell [PJGF96]. Concurrent Haskell allows to fork new threads with the primitive `forkIO` and thread communication through one-place channels called `MVar`s.

Since we want to enumerate all the `One` leaves of a given `SearchTree`, we have to start at the root and walk through the tree. The idea is that whenever a `Choice` node is encountered, a new `MVar` is created and a new thread is forked. The new thread processes the left branch, the old one does the same for the right one. When they are finished, the new thread writes its result in the `MVar`. The old thread takes this result out of the `MVar`, combines it with its own result and returns the combined results.

So far the performance of this search would be poor in most cases. To fork a new thread for every `Choice` node would create too much overhead. To reduce the overhead, we only fork threads up to a certain depth and then use a sequential search for the subtrees.

In order to use all cores, the depth should be at least $\lceil \log_2 n \rceil$, where $n$ is the number of cores. However, the optimal depth depends on the shape of the tree. If one of the branches is relatively small, a thread will finish early or some threads are not forked at all. Since the shape of the tree is not easily observable, a good heuristic is to increase the above mentioned minimum by one. Our experiments have shown that the performance suffers only little by the increased scheduling overhead, but in some cases the additional threads will help to keep the processors busy.

This approach works best if the tree that is searched is almost balanced. If that is the case there are considerable speedups. If the tree is degenerated, the search performs only little worse than the sequential search. This is due to the fact that the parallelization overhead is constant for a fixed depth.

In a benchmark of an early version of this search, we noticed that the program hardly used more than one core. The problem was that the results of the sequential searches is not demanded during the parallel search. Since Haskell is a lazy language this means that the threads don't evaluate the sequential search at all. Only when the result of our parallel search is demanded by another thread, this thread will actually execute the sequential searches. To solve this problem, we had to force the complete evaluation of the sequential searches to ensure that the work is done by the forked threads as intended.

By making the sequential searches strict another problem arises: The search only works for finite trees. That limits this approach to search for results for problems with finite search spaces. There are problems that are best solved with infinite search spaces, for example test data generation, where this approach is too limited.

### 3.2 Second Approach: Bag of Tasks

Our second approach was to use a technique known as Bag of Tasks [And99]. There is a data structure that holds a set of tasks, the bag, and there is a gang of threads, the workers. Every worker takes a task out of the bag, processes it and either puts new tasks back into the bag or returns a result.

We use buffered channels for both, the bag and the results. Initially, we put the tree we want to search in the bag. A thread of the gang (ideally the gang consists of as many threads as there are cores[2]) takes the tree out of the bag. If the root of the tree is a `Choice` the thread puts the two subtrees back into the bag and continues. If the root is a `One` the value is written into the result channel and the thread continues. At last, if the root is a `None`, the thread just continues.

The channel we use to return the results allows to obtain its contents as a lazy list. This means that it is possible to process infinite trees when only a finite number of results are actually needed.

Since the order of the search almost resembles breath-first search, this search is complete. On the other hand, the worst case space complexity is exponential.

An advantage of this approach is that it is unlikely for threads to run out of work because new tasks are created frequently. On the other hand, such frequent generation of new tasks may affect the performance negatively because of synchronization overhead. There are at least two accesses to a shared data structure for every node in the tree: one to put it into the bag, one to take it out of the bag and, if necessary, one to put the result to the result channel. Nevertheless this approach performs well as long as the time spent accessing the shared data structures is negligible compared to the time needed to evaluate a node in the tree.

The algorithm works well with infinite trees but now there is a problem with finite trees. A thread processing the list of results is not able to tell if all results have been found. For example the attempt to count all the results will not terminate, even if the tree is finite.

To solve this problem we need to detect if all tasks in the bag have been processed. To achieve this we use an additional shared data structure, a counter. The counter represents the number of tasks that have not been processed. When a thread finishes to process a node, the counter is decreased, when a new node is added to the bag the counter is increased. If a thread that has finished the processing of a node decreases the counter to zero, there are no more tasks to process and a symbol is written to the result channel which indicates that there are no more values to read.

[2] Due to scheduling issues of the OS it may be better to have one thread less than cores. See [MJS09].

### 3.3 Third Approach: Semi-explicit Parallelism

Our third approach is the use of Glasgow parallel Haskell (GpH) which provides semi-explicit parallelism [THLP00] via a new combinator:

```
par :: a -> b -> b
```

The combinator `par` just returns its second argument but additionally stores its first argument as a so called *spark* in a *spark pool*. Idle processors can find it there and evaluate it to weak head normal form. A programmer can add the `par` combinator whenever she thinks it might be beneficial to evaluate an expression in parallel that is probably needed in the rest of the program.

Applied to our problem, the resulting program is quite simple:

```
search :: SearchTree a -> [a]
search None         = []
search (One x)      = [x]
search (Choice l r) = rs `par` (search l ++ rs)
 where rs = search r
```

As soon as the first element of the result list is demanded, the `par` combinator causes all right branches on the path down to the first result to be sparked. Eventually an idle processor will evaluate such a spark and thereby creates more sparks and so on, until the whole tree has been evaluated. If the tree is infinite and only a finite number of the results are demanded, the remaining sparks will eventually be removed by the garbage collector.

The performance of the parallel part mainly depends on the run-time system. According to [MJS09], the improvement of the run-time system is work in progress.

The use of the append operator `(++)` makes the worst case execution time of this approach quadratic in the number of results. The benchmarks we ran had only a small number of results so this didn't affect the measured times.

Note that this version of the search returns the results in depth-first search order. Thus, in contrast to our second approach, this search is not complete.

## 4 An Example for a Non-deterministic Application

To avoid benchmarking with the usual artificial toy applications, we opted for using a monadic SAT solver[3]. The SAT solver implements the Davis-Putnam-Logemann-Loveland algorithm [DLL62]. It non-deterministically assigns a boolean value to a variable and then tries to simplify the formula. The results are all assignments to the variables that satisfy the formula. Note that this implementation was not designed to compete with efficient

[3] Available from Hackage as package incremental-sat-solver [inc]

| | | number of cores | | | |
|---|---|---|---|---|---|
| | | N1 | N2 | N3 | N4 |
| programs | fork to depth | 302s | 176s | 117s | 119s |
| | bag of tasks | 307s | 172s | 130s | 114s |
| | gph | 269s | 137s | 99s | 83s |
| | list | 264s | 266s | 287s | 298s |

Figure 2: Wall-clock results

SAT solvers. Nevertheless, it serves as a reasonable benchmark for our search algorithms applied to expensive non-deterministic computations.

The implementation of the SAT solver uses the monadic operations described in section 2 so no changes to its implementation were needed to apply our searches.

Then there are two more advantages to use this application for our benchmarks: The search space is finite which is necessary for our first approach. And to compute a single node in the tree needs a non-negligible amount of time, i.e., our second approach should perform well.

## 5 Benchmark Results

We tested our searches with boolean formulas from [SAT]. All benchmarks were made on a Intel Core2 Quad CPU with 2.83 GHz using a snapshot version of ghc 6.11 that already benefits from some of the results given in [MJS09]. Although the machine runs a 64 bit Ubuntu 8.10, we ran our benchmarks using 32 bit binaries since, according to [MJS09], 64 bit programs tend to need more garbage collection resulting in less parallelism. To decrease garbage collection further, we ran all tests with 1 GB of initial heap.

Figure 2 gives the average wall-clock time needed to prove the unsatisfiability of a 3-SAT instance with 150 variables and 645 clauses. The rows correspond to the three different approaches presented in this paper and the list monad without any parallelization. The columns correspond to the number of cores we used. Figure 3 shows the same data as a graph. The results of other boolean formulas, also satisfiable ones with more than 2000 solutions, are comparable.

Note that when we use four cores, the speedup compared to three cores is very small or the performance is even worse than with three cores. This can be explained by an effect described in [MJS09]. The problem is that when all cores are used one or more threads are frequently descheduled by the OS for a relatively long period. When one of the threads initiates garbage collection, all other threads need to stop. If one of the threads is descheduled, all other threads have to wait until it is rescheduled again.

The optimal result would be, that when using $n$ cores, the runtime is $\frac{1}{n}$th of the runtime of one core. It is easy to see that we don't meet that requirement. But of course there is some

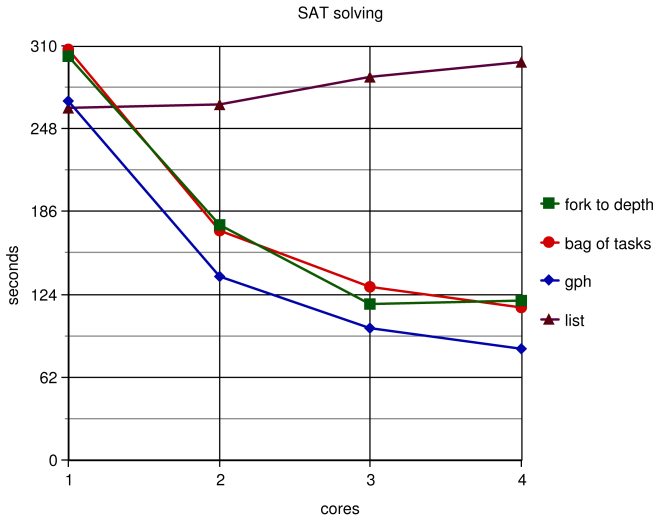


Figure 3: Graph of the wall-clock results

overhead, e.g., to combine the results of parallel computations and, not at least, garbage collection takes some time, too. So, we consider our results quite satisfactory, at least for this benchmark.

The comparison with the list monad, i.e., without any parallelization, shows that when using only one processor, the parallel algorithms are not significantly slower. However, when we use two or more processors, the speedup w.r.t. the list monad is significant. We expected that our implementation with the list monad shows the same runtime using one or more processors. The actual drop in performance is possibly due to the fact that when we use more than one core, the garbage collector runs in parallel resulting in more overhead.

A characteristic of this benchmark, that suits our approaches good, is that it is not trivial to compute a node in the search tree. The more time is spend to evaluate a node in the tree, the less significant is the synchronization overhead and conflicting accesses to shared data structures are less likely.

In contrast remember the example in Section 2 that shows how to nondeterministically compute a permutation of a list. To evaluate a node in the tree produced by a call of `perm xs` hardly takes any time. To compute the number of permutations of the list `[1..10]` using the list monad only takes half the time of the fastest of our parallel searches. The bag-of-tasks approach even performs 20 to 50 times worse than the list. To use more cores only results in more synchronization overhead and, hence, in worse performance.

We consider it promising that the benchmark closest to a real world application, i.e. the SAT solver, showed the best gain in performance.

## 6 Related Work

There are many approaches to the parallelization of declarative languages, we only mention a few.

The Eden language [LOMPM05] is a parallel functional language based on Haskell. It introduces the concept of processes for parallel execution. In contrast to our work, it extends the Glasgow Haskell Compiler, and gives the programmer more control over the evaluation.

Several Prolog compilers offer low-level predicates for multi-threading support. Recently an experimental implementation of high-level multi threading for the object-oriented logic programming language Logtalk [MCN08] has been presented.

## 7 Future Work

In [AB07] Sergio Antoy and Bernd Braßel motivated the need of a primitive that provides access to all results of a non-deterministic computation in Curry. More work on this topic has been done recently by Antoy and Hanus [AH09].

Since there is a Compiler that translates Curry to Haskell, namely KiCS [KiC], it should be possible to use our results to provide a similar primitive to search for all results of a non-deterministic computation in parallel.

Furthermore, we still work on the improvement of our searches. A concrete plan is to use Glasgow parallel Haskell to implement a complete search.

## 8 Conclusion

We presented three approaches to parallelize search for results of non-deterministic computations. We use a tree to represent the computation.

Our first approach was to divide subtrees of the tree between multiple threads. This approach only works for finite trees and works best when the tree is balanced.

The second approach was to use a technique known as Bag of Tasks. A gang of threads repeatedly process tasks from the bag. This approach also works for infinite trees and implements a complete search. Since there is much communication overhead, this approach scales only if it is expensive to compute nodes of the search tree.

Finally, the third approach used semi-explicit parallelism but the search it implements is not complete.

With all three approaches we achieved serious speedups for a non-trivial non-artificial problem.

Our third approach seems to be the best choice, at least if the search space is finite. Fur-

thermore, this parallel implementation of depth-first search was by far the easiest to implement.

We believe that our work will eventually help to add parallel evaluation to the functional logic language Curry. We already work on a compiler that translates Curry to Haskell using a monadic style. We hope to be able to seamlessly integrate parallel search to this translation.

## References


[AB07] Sergio Antoy and Bernd Braßel. Computing with subspaces. In *PPDP '07: Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 121–130, New York, NY, USA, 2007. ACM.

[AH09] S. Antoy and M. Hanus. Set Functions for Functional Logic Programming. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09)*. To appear in ACM Press, 2009.

[And99] Greg R Andrews. *Foundations of Parallel and Distributed Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Cur] Curry. `http://curry-language.org`. Online; 24.04.2009.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[FKS09] Sebastian Fischer, Oleg Kiselyov, and Chung-chie Shan. Purely Functional Nondeterministic Programming. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional Programming*. ACM, 2009. To appear.

[Has] Haskell. `http://haskell.org`. Online; 24.04.2009.

[inc] incremental-sat-solver. `http://hackage.haskell.org/cgi-bin/hackage-scripts/package/incremental-sat-solver-0.1.7`. Online; 24.04.2009.

[KiC] KiCS. `http://www.informatik.uni-kiel.de/prog/mitarbeiter/bernd-brassel/projects`. Online; 24.04.2009.

[LOMPM05] Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña-Marí. Parallel functional programming in Eden. *J. Funct. Program.*, 15(3):431–475, 2005.

[MCN08] Paulo Moura, Paul Crocker, and Paulo Nunes. High-Level Multi-threading Programming in Logtalk. In *Practical Aspects of Declarative Languages, 10th International Symposium, PADL 2008*, pages 265–281, 2008.

[MJS09] Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime Support for Multicore Haskell. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional Programming*. ACM, 2009. To appear.

[PJGF96] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *23rd ACM Symposium on Principles of Programming Languages*, pages 295–308, St Petersburg Beach, Florida, January 1996. ACM.

[SAT] SATLIB. http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html. Online; 24.04.2009.

[THLP00] P. W. Trinder, K. Hammond, H. W. Loidl, and Peyton. Algorithm + strategy = parallelism. *Journal of Functional Programming*, 8(01):23–60, 2000.

[tre] tree-monad. http://hackage.haskell.org/cgi-bin/hackage-scripts/package/tree-monad-0.2. Online; 24.04.2009.

[Wad92] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14. ACM Press, 1992.