Integration of Parallel and Fair Search Strategies for Non-Deterministic Programs into the Curry System KiCS2

Bastian Holst

Master's thesis
submitted in May 2014

Christian-Albrechts-Universität zu Kiel
Institut für Informatik
Arbeitsgruppe für Programmiersprachen und Übersetzerkonstruktion

Advised by: Prof. Dr. Michael Hanus

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kiel, 27.05.2014

_____

Abstract

Evaluating non-deterministic expressions in functional logic programming languages like Curry can be implemented as a search on binary search trees. The Curry implementation KiCS2 is written in Haskell and allows the definition of various search strategies on such trees. In this thesis, we integrate various parallel search strategies in KiCS2, which are based on depth-first search and breadth-first search as well as a different search technique, which is more complete compared to breadth-first search.

We use three approaches of parallelism: using semi-explicit parallelism, using a bag of tasks approach, and starting individual computation threads for each branch of the search tree. We implement these approaches in multiple ways and discuss their advantages and limitations, especially in terms of resource consumption. In particular, we present various approaches limiting the communication overhead and memory consumption. In order to confirm or refute our considerations, we perform benchmarks using multiple example programs.

Contents

1. Introduction **6**
- 1.1. Notation of Source Code . . . 7
- 1.2. Motivation . . . 7
- 1.3. Outline . . . 8

2. Technologies **9**
- 2.1. Functional Logic Programming with Curry . . . 9
  - 2.1.1. Expressions . . . 9
  - 2.1.2. Types . . . 10
  - 2.1.3. Global Definitions . . . 11
  - 2.1.4. Constraints . . . 12
  - 2.1.5. Call-Time Choice Semantics . . . 13
  - 2.1.6. Set Functions . . . 14
- 2.2. Haskell . . . 15
  - 2.2.1. Semi-Explicit Parallelism . . . 17
  - 2.2.2. Concurrent Haskell . . . 19
  - 2.2.3. Runtime System . . . 25
- 2.3. KiCS2 . . . 25
  - 2.3.1. Representing Non-Determinism in Data Structures . . . 25
  - 2.3.2. Search Tree . . . 28

3. Evaluation Criteria **30**
- 3.1. Resource Consumption . . . 30
- 3.2. Completeness . . . 30
- 3.3. Stopping the Computation . . . 32

4. Search Strategies **33**
- 4.1. Search Strategies as Part of the KiCS2 System . . . 33
- 4.2. Sequential Search Strategies . . . 35
  - 4.2.1. Depth-First Search . . . 35
  - 4.2.2. Breadth-First Search . . . 36
- 4.3. Order-Preserving Parallel Search Strategies . . . 37
  - 4.3.1. Original Approach to Deterministic Parallel Depth-First Search . . . 37
  - 4.3.2. Deterministic Parallel Depth-First Search with Strategies . . . 38
  - 4.3.3. Reducing the Number of Sparks for Parallel Depth-First Search . . . 40
  - 4.3.4. Deterministic Parallel Breadth-First Search with Strategies . . . 44

4.4. Bag of Tasks . . . 47
4.4.1. General idea . . . 48
4.4.2. Depth-First Search . . . 49
4.4.3. Breadth-First Search . . . 51
4.5. Fair Search . . . 53
4.5.1. Primitive Fair Search . . . 53
4.5.2. Fair Search with Chained Threads . . . 56
4.5.3. Using Exceptions for Communication between Search Threads . . . 61

5. Technical Details **64**
5.1. Stopping Parallel Evaluation . . . 64
5.1.1. Stopping Threads Explicitly . . . 65
5.1.2. Using Finalisers on Weak Pointers . . . 66
5.2. Bag of Tasks Implementation . . . 68

6. Evaluation **74**
6.1. Completeness . . . 74
6.2. Abortion Behaviour . . . 75
6.3. Performance Analysis . . . 77
6.3.1. Benchmark Programs . . . 77
6.3.2. Benchmarking System . . . 83
6.3.3. Results . . . 83
6.4. Summary . . . 106

7. Usage of Parallel Search **108**

8. Future Work **111**
8.1. Manual and Automatic Annotation . . . 111
8.2. Prolog's AND-Parallelism . . . 112
8.3. Parallel Iterative Deepening Search . . . 114

9. Conclusion **116**

Index **118**

Bibliography **118**

A. Wrong Divisor Implementation as an Application of Fair Search Strategies **121**

B. Benchmark Results of a Selection of Search Strategies **122**

C. Contents of the Data Medium **128**

1. Introduction

Improving the performance of single processor cores has become harder and harder in the last years. Instead, hardware developers chose to add more and more cores to single processors. As a consequence, parallel programming is becoming more relevant.

However, writing parallel programs is still a challenge. Implementing parallelism manually based on threads and locking is often expensive and error-prone. Using transactional memory systems like *software transactional memory* in Haskell [11] reduces the difficulty of implementing parallelism, but it is primarily targeted at concurrent programming. Additionally, functional programming languages offer primitives allowing the programmer to write parallel programs without reasoning about threads and locking [31].

The referential transparency of declarative programming languages encourages relying on the compiler and the runtime system. Unfortunately, this has not been proven to be fruitful for large-scale functional programs [31]. Non-deterministic programs benefit from this more, which has been shown for the logic programming language Prolog. There are two types of parallelism exploited by Prolog implementations: *AND-parallelism* [6] and *OR-parallelism* [32]. AND-parallelism means resolving multiple subgoals for one result in parallel whereas OR-parallelism is the parallel computation of multiple results in a non deterministic computation.

The programming language *Curry* [10] combines aspects of functional and logic programming, so it might be reasonable to try parallelisation techniques from functional and from logic programming. Nevertheless, this thesis is focused on one of the parallelisation techniques known from logic programming: OR-parallelism.

The Curry system *KiCS2* implements Curry's non-determinism in the functional programming language *Haskell* [4]. In KiCS2 it is possible to choose between multiple search strategies. Reck and Fischer [30] presented a first approach towards parallel search for non-deterministic solutions in Haskell in 2009. While Reck and Fischer concentrated on strategies resulting in a better performance, it is also possible to implement fair search strategies in parallel. Fair search strategies allow the computation of results even if the program contains a deterministic loop in a branch.

For this thesis, we integrated various search strategies into KiCS2 and compared their behaviour using various benchmark programs. These integrated search strategies can be used as a top-level search strategy to automatically use parallel evaluation on the whole program but also to be used inside a complete program to parallelise certain expensive computations.

1.1. Notation of Source Code

The code examples in this paper use the functional language Haskell [20] and the functional logic language Curry [10]. The syntax of Haskell and Curry programs is quite similar, so it may be hard to distinguish Curry from Haskell programs. Unless noted otherwise, search strategies are implemented in Haskell whereas program examples and examples of usage are written in Curry.

Both Haskell and Curry code is layouted using *lhs2TeX*[1] and thus contains beautifications such as arrows $\rightarrow$ (`->`) and $\leftarrow$ (`<-`), lambda symbols λ (`\`), and certain relation symbols $\neq$ (`/=`), $\equiv$ (`==`), $\leq$ (`<=`), $\geq$ (`>=`), and $\wedge$ (`&&`).

1.2. Motivation

While other Curry implementations only offer one search strategy, in KiCS2 the user can choose between multiple search strategies. The parallel search strategies, implemented as part of this master's thesis, may be selected as a top-level search strategy. To use a parallel search strategy, the user first sets the number of system threads; for example when using a 12 processor system, the user sets the number of system threads to 12. After that, it is possible to enable parallel evaluation according to the selected strategy.

```
:set threads 12
:set +parallel
```

Furthermore, it is possible to use parallel search strategies in a program to evaluate only parts of it in parallel. Therefore, we made parallel search strategies available as a Curry library module. This module contains the parallel search strategies and the functions *getAllValues* and *getOneValue*.

```
getAllValues :: Strategy → a → IO [a]
getOneValue :: Strategy → a → IO (Maybe a)
```

Both functions take a strategy and an arbitrary expression. *getAllValues* returns all values of the expression and *getOneValues* returns *Nothing* if it has no values and otherwise any of the values. The following example program computes all values of the expression *e* and prints the minimum. *splitAll* is one of the parallel search strategies.

```
main = do
  results ← getAllValues splitAll e
  let m = minimum results
  putStrLn ("Minimum: " ++ show m)
```

[1] `http://www.andres-loeh.de/lhs2tex/` [accessed 25-May-2014] by Andres Löh

1.3. Outline

This section gives a short outline of the remainder of this thesis. After this introduction, chapter 2 gives a short insight in the technologies used throughout this thesis. These technologies include the functional logic programming language Curry, which is explained in section 2.1, and the parallelisation features of the functional programming language Haskell used for the implementation of search strategies, section 2.2. Furthermore, we outline the basic implementation of the Curry system KiCS2 in section 2.3.

Chapter 3 then gives an overview on the requirements we have on parallel search strategies. This can be used as a foundation for the chapter 4, which shows the implementation of the search strategies and how these are integrated in KiCS2.

Chapter 5 then presents various details of the implementation. These are not necessarily needed to understand the remainder of the thesis, but give an overview of technical obstacles, which occurred during the implementation.

Having explained the major parts of the implementation, the following chapter 6 contains the comparison and evaluation of the search strategies. Besides presenting benchmark results in section 6.3, we talk about to which degree other requirements are held.

After the evaluation, chapter 7 contains further information on how to use the parallel search strategies in a Curry program. Chapter 8 then introduces possible other ways of parallelisation and possible improvements of the given strategies. Finally, chapter 9 concludes the results of this thesis.

2. Technologies

Before starting to implement parallel search strategies for the functional logic programming language Curry, we have to lay some foundations. As a part of this, section 2.1 first gives a short introduction into Curry. To implement parallel search strategies, we make use of Haskell's concurrency and parallelisation features. The section 2.2 gives a short introduction on these. Furthermore, the basic implementation of non-determinism in KiCS2 is explained in section 2.3.

2.1. Functional Logic Programming with Curry

Curry is a multi-paradigm language combining important features of functional languages with features of logic languages in a single language. In particular, it provides functional programming features like higher-order functions, parametric polymorphism and demand-driven evaluation and features from logic programming languages like computing with built-in search, free variables, and partial data structures as well as computing with constraints [10]. In fact, Curry is in many ways similar to the functional programming language Haskell, which is now widely used in academics but also in the industry.

A Curry program consists of the definition of data types and operations on these types. The syntax of Curry resembles the syntax of the functional programming language Haskell. Names of variables and operations usually start with a lowercase letter whereas type constructors and constructors start with an uppercase letter.

2.1.1. Expressions

An important component of Curry programs are *expressions*. Expressions are:

- basic *values* like numbers (−2, 3.141),
- an *application* of an operation f to an expression ε written as the juxtaposition $f\ \varepsilon$,
- an application of an *infix operator*, for example the arithmetic operators + or $*$ (42 + 1, $2 * 1$),
- a *data constructor* C applied to other expressions $\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n$ written as the juxtaposition $C\ \varepsilon_1\ \varepsilon_2 \ldots \varepsilon_n$ (*True*, *Nothing*, *Just* 2),
- *conditional* expressions **if** ε_b **then** ε_1 **else** ε_2,

- *let* expressions **let** $\{x_1 = \varepsilon_1; x_2 = \varepsilon_2; ...; x_n = \varepsilon_n\}$ **in** ε, where $\varepsilon, \varepsilon_1, \varepsilon_2, ..., \varepsilon_n$ are expressions,
- *functions*, which can be written analogous to the λ-Calculus [8] as $(\lambda x \longrightarrow \varepsilon)$, where ε has to be an expression.

2.1.2. Types

Curry is statically typed with a type system inspired by the type system presented by Milner in 1978 [27]. Its type system is, therefore, very similar to that of Haskell. Each expression in the program has an unique type which, in most cases, is inferred automatically by the compiler. The programmer can provide type signatures, but in general he does not have to. It features parametric polymorphism as well as type inference at compile time. Currently there are attempts to extend Curry's type system by type classes [5], designed for functional programming languages by Wadler and Blott in 1989 [34]. Several types are predefined, but it is also possible to define data types yourself.

Among the predefined types are types for *numbers*, *boolean values*, *tuples*, *lists*, *characters*, *strings*, *operations*, and *constraints*.
The type for integers is called *Int*. Integral numbers are constructed by values like 42 or −15. Typical operators for integers are +, −, and ∗, which are evaluated only when both arguments have been evaluated to values. Otherwise the function calls are suspended. The type for floating point numbers is called *Float* and its constructors are values like 3.14159 and −5.0e−4. The arithmetic functions are called differently compared to those on integers (+., −., ∗., /.), but they have a similar behaviour.

In addition to predefined data types, data types can be declared in the following form. τ_{ij} are types which may itself contain the type parameters α_1 to α_n and C_1 to C_k are the constructors of the newly defined type.

$$\textbf{data}\ T\ \alpha_1 ... \alpha_n = C_1\ \tau_{11} ... \tau_{1n_1} \mid ... \mid C_k\ \tau_{k1} ... \tau_{kn_k}$$

Although the type for boolean values is predefined, it could be defined in the following form.

$$\textbf{data}\ Bool = True \mid False$$

The definition can be read as: A boolean can be *True* or *False*. Consider the definition of two other commonly used types.

$$\textbf{data}\ Maybe\ a\ \ = Nothing \mid Just\ a$$
$$\textbf{data}\ Either\ a\ b = Left\ a\ \ \mid Right\ b$$

In contrast to *Bool*, *Maybe* and *Either* are not types standing for its own, *Maybe* and *Either* are *type constructors*. A type constructor has to be applied to one or multiple types to get a type. An application of a type constructor T to a type τ is written as the juxtaposition $T\ \tau$. The application of a type constructor to a type forms a type (like *Maybe Bool*) or another type

constructor (like *Either Int*) which has to be applied to another type to instantiate a type (for example *Either Int Float*).
The definition of *Maybe* can be read as: *Nothing* is a value of type *Maybe* τ and, given x is of type τ, *Just* x is a value of type *Maybe* τ. *Maybe* is a type often used for optional parameters or functions which return values only under certain conditions.

Lists of values of type τ have the type $[\tau]$. As a result, $[\cdot]$ is a type constructor as well. The constructors of lists are [ ], which is an empty list, and the infix operator :, which prepends a value to an existing list. Let x be an expression of the type τ and xs a list of values of type τ, then $x : xs$ is a non-empty list of type $[\tau]$. The list $x_1 : x_2 : \ldots : [\,]$ can also be written in the convenient notation $[x_1, x_2, \ldots]$. A type, of which values are often stored in lists, are characters. Characters like `'a'` or `'9'` are constructors of the type *Char*. *Strings* are represented as lists of characters and thus can be written as `'H' : 'e' : 'l' : 'l' : 'o' : []` or `['H', 'e', 'l', 'l', 'o']`, but there is also the more convenient notation `"Hello"`. In addition to lists, tuples are available to structure data. If $\tau_1, \tau_2, \ldots, \tau_n$ are types and $n \geq 2$, then $(\tau_1, \tau_2, \ldots, \tau_n)$ is the type of all n-tuples. Let x_i be an element of type τ_i for $i \in \{1, 2, \ldots, n\}$ then $(x_1, x_2, \ldots, x_n)$ is an element of type $(\tau_1, \tau_2, \ldots, \tau_n)$. The unit type () has only a single element () and can be interpreted as the type of 0-tuples.

Another type is the type of functions or operations $\tau_1 \longrightarrow \tau_2$, where τ_1 and τ_2 are types. For example the type of the operator + is $Int \longrightarrow Int$. A function of type $\tau_1 \longrightarrow \tau_2$ can be written analogous to the λ-Calculus [8] as $(\lambda x \longrightarrow e)$, while x is a parameter of type τ_1 and e is an expression of τ_2.

The types introduced so far are all available in Haskell as well showing the similarity between both type systems. But one type specific to and characteristic for Curry is *Success*.

data *Success* = *Success*

This is the type of successful evaluation. Although the type *Success* is similar to the previously described unit type as it has only one value, both are different in regard to their use. Expressions of the type *Success* are used as conditions for defining rules and are also called *constraints*.

2.1.3. Global Definitions

An *operation* f with the formal parameters $x_1, \ldots, x_n$ defined by the expression e is written as $f\ x_1 \ldots x_n = e$. For example we can define the *square* operation as:

$square\ x = x * x$

A *constant* is an operation with zero parameters and can be defined in a similar way.

$answer = 42$

In functional logic languages expressions can yield zero, one, or multiple results. A simple example for an operation yielding multiple results is *coin*, the operation simulating a coin flip.

$coin = 0$
$coin = 1$

This example reveals an important semantic difference compared to Haskell. In Haskell the operation *coin* will always yield the result of its first rule 0 whereas in Curry it non-deterministically yields both values: 0 and 1.

Another non-deterministic operation is the predefined infix operator ?, called *choice*, which may be defined as:

$x \,?\, \_ = x$
$\_ \,?\, y = y$

According to the definition, the expression 0 ? 1 has two values: 0 and 1, similar to the operation *coin*.

2.1.4. Constraints

Given the possibility of multiple values from one expression, we often want to select specific values. This can be done by constrained operation definitions. An operation definition is constrained by the constraint c with $f\ x_1 \ldots x_n \mid c = \varepsilon$. The constraint c is an expression with the type *Success*.

An elementary constraint is the *strict equality* =:=. The expression $\varepsilon_1 =:= \varepsilon_2$ yields *Success* if and only if both ε_1 and ε_2 can be evaluated to the same ground data term. If one or both sides is non-terminating, the strict equality would not hold and the evaluation of $\varepsilon_1 =:= \varepsilon_2$ would also be non-terminating. Using this equality constraint we can define a simple definition of the function *last* which returns the last element of a given list.

$List\ a = [\,] \mid a : (List\ a)$

$append :: [a] \to [a] \to [a]$
$append\ [\,]\ \ \ ys = ys$
$append\ (x : xs)\ ys = x : (append\ xs\ ys)$

$last :: [a] \to a$
$last\ zs \mid append\ xs\ [x] =:= zs = x$ **where** x, xs **free**

In addition to the constraint, the definition of *last* makes use of *free variables*. The Curry system searches in all possible values for the variables x and xs and then returns the value for the variable x.

2.1.5. Call-Time Choice Semantics

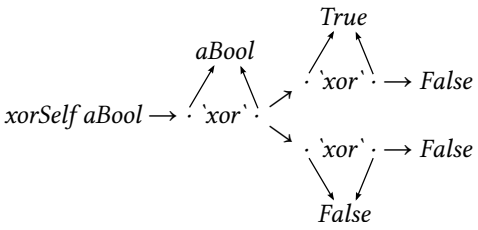
Using the choice operator, we can define the non-deterministic operation *aBool*, the exclusive disjunction *xor* on boolean values, and *xorSelf* [4]:

$$
\begin{aligned}
&aBool = True \;?\; False\\
&True \;\grave{}xor\acute{}\; True \;= False\\
&True \;\grave{}xor\acute{}\; False = True\\
&False\;\grave{}xor\acute{}\; x \quad\;\; = x\\
&xorSelf\;\; x = x\;\grave{}xor\acute{}\;x
\end{aligned}
$$

Now consider the expression *xorSelf aBool*. Interpreting the program as a term rewriting system, we could have the following reduction:

$$
\begin{aligned}
xorSelf\;\; aBool &\longrightarrow aBool\;\grave{}xor\acute{}\; aBool \longrightarrow True\;\grave{}xor\acute{}\; aBool\\
&\longrightarrow True\;\; \grave{}xor\acute{}\; False \longrightarrow True
\end{aligned}
$$

The given outermost reduction results in the unintended result *True*, which would not be possible using a strict evaluation strategy. To exclude such unintended results, Curry does not allow this reduction by using a *call-time choice* semantics [13]: the values of non-deterministic expressions are determined at the time of the application of an operation. This results in the same reductions as strict evaluation, but it does not require an eager evaluation of the arguments. In fact there is a lazy evaluation strategy for functional logic languages with call-time choice semantics using sharing between the arguments of an operation [1]. Thus, we can assume that we can evaluate this expression lazily with the occurrences of *aBool* shared after the application of *xorSelf*, so that both occurrences of *aBool* are either evaluated to *True* or *False*. Thus, the expression only has the expected result *False*. The evaluation of the above expression can be visualised in the following way:



```
data Person = Ada | Bernhard | Camelia | Kofi | Husain | Hikari | Rayen | Gaurav

mother Rayen   = Hikari
mother Gaurav  = Hikari
mother Hikari  = Ada
mother Kofi    = Ada

father Rayen   = Husain
father Gaurav  = Husain
father Hikari  = Bernhard
father Kofi    = Bernhard

parent x = father x
parent x = mother x

child x | parent y =:= x = y where y free
```

Figure 2.1.: A variant of the classic family relations program in Curry.

2.1.6. Set Functions

Figure 2.1 shows a variant of the classic family tree example. The data type *Person* defines a set of people which can be the *mother* or the *father* of a child. Furthermore, we define the operation *parent*, which denotes non-deterministically the parents of a child. In the definition of the operation *child* we make use of free variables to invert the definition of *parent*. Note that *child* may also yield multiple values non-deterministically.

With this definition it is possible to define the predicate *hasChild* in the following way:

```
hasChild x | child x =:= _ = success
```

A person has a child if there is a child of this person. The name of the child is a free variable like y in the operation *child*, but it was omitted as the name of the child is not needed.

The operation *hasChild* works as expected: the evaluation of *hasChild Camelia* fails and the evaluation of *hasChild Husain* evaluates to *Success*. In fact, *hasChild Husain* evaluates to *Success* in two ways: one time through the child *Gaurav* and one time with the help of *Rayen*. As a result, we get the value *Success* two times. This may be undesired; we may just like two know if there is any child of *Husain*.

We can use set functions [2] in this case. For any operation f, f_S is called its set function. f_S is used to compute a set of values given by f. Using this concept, we capture the non-determinism of the function f, but not the non-determinism origination from its arguments. In Curry, we can get the set function of *isChild* with the operation *set*1 as *isChild* has one argument. There are corresponding set operations for other numbers of arguments.

```
set1 :: (a1 -> b) -> a1 -> Values b
```

The result type *Values b* is a multiset of the result values. As it has an arbitrary order, there are some operations on it, which do not depend on its order.

$$\begin{aligned} &\mathit{isEmpty} &&:: \mathit{Values}\ \_ \to \mathit{Bool} \\ &\mathit{sortValues} &&:: \mathit{Values}\ a \to [a] \end{aligned}$$

The operation *isEmpty* checks whether the set of values is empty and the operation *sortValues* sorts all values of the set in ascending order.

With the help of *isEmpty*, we can define *hasChild* again. A person has a child if and only if the set of children is not empty.

$$\mathit{hasChild}\ x \mid \neg\ (\mathit{isEmpty}\ ((\mathit{set1}\ \mathit{child})\ x)) = \mathit{success}$$

With this definition of *hasChild*, the evaluation of *hasChild Husain* only yields one *Success* as expected.

The call of *hasChild* (*Husain* ? *Hikari*) still yields two values, because the non-determinism originating in the arguments of the set function is not captured and both *Husain* and *Hikari* have children.

2.2. Haskell

The Curry compiler KiCS2 compiles the functional logic language Curry (section 2.1) to Haskell reusing the given implementations of demand-driven evaluation and higher-order functions. The implementation of the non-deterministic operations is explained in section 2.3. As Haskell is widely known in academics and the syntax of Haskell is similar to that of Curry, an introductive characterisation of the language is omitted here. Instead, this section describes certain library modules, which have to be understood to implement various parallel search strategies. Additionally, it describes some specifics related to the runtime system of the *Glasgow Haskell Compiler* (*GHC*) [23]. The GHC is the most important Haskell implementation and it is also used by KiCS2 to compile the generated Haskell code. Readers looking for a definition of Haskell are encouraged to read the *Haskell 2010 Report* [20] whereas readers looking for a gentle introduction to Haskell may consider the book *Learn You a Haskell for Great Good* [16].

Reachability

The runtime system of the Glasgow Haskell Compiler includes a *garbage collector* to free memory occupied by objects that are not longer accessible from the program. GHC's garbage collector is a tracing garbage collector and thus determines which objects are *reachable*. A reachable object is, informally defined, an object that can still be accessed from the program. More formally, an object is reachable if and only if at least one of the following statements is true:

- The object is in the *root set*. The root set is a set of objects which are assumed to be reachable. In Haskell, these are all global objects and objects, which are referenced in the call stack of each Haskell thread as a local variable or function argument.
- The object is referenced from another object that is reachable.

All objects which are not reachable are *garbage* and the memory they occupy can be recovered. The implementation of the garbage collector in GHC is described by Marlow et al. [7, 21, 22].

Weak References

Sometimes it is necessary to hold a reference to an object without keeping it *reachable*. The standard solution for this problem are *weak references* [15]. Consider the following interface.

$$\textbf{data}\ \mathit{Weak}\ v$$
$$\mathit{mkWeakPtr} :: v \longrightarrow \mathit{Maybe}\ (\mathit{IO}\ ()) \longrightarrow \mathit{IO}\ (\mathit{Weak}\ v)$$
$$\mathit{deRefWeak} :: \mathit{Weak}\ v \longrightarrow \mathit{IO}\ (\mathit{Maybe}\ v)$$

The function *mkWeakPtr* creates a weak reference to the object given as the first argument. The *IO* action provided as the second argument is an optional *finaliser*. For as long as this object is reachable by other means than through weak references, the function *deRefWeak* will return this object. When this object is not reachable anymore, it may be *finalised* by the garbage collector. Finalizing means making *deRefWeak* called on weak reference to this object return *Nothing* and then run the finaliser. After the finalisation, the object may be discarded. Note that the reachability of the weak reference object does not affect the reachability of its value and neither a reference in the finaliser nor in the weak reference itself keeps the value reachable.

Haskell's weak pointers are in fact *key/value weak references* and therefore have an even more general constructor.

$$\mathit{mkWeak} :: k \longrightarrow v \longrightarrow \mathit{Maybe}\ (\mathit{IO}\ ()) \longrightarrow \mathit{IO}\ (\mathit{Weak}\ v)$$

The function *mkWeak* takes, besides the finaliser, a *key* of type k and a *value* of type v and creates a weak pointer of type $\mathit{Weak}\ v$. This means for the reachability that the value of the weak reference is reachable if the key of the weak reference is reachable[1]. Again, neither the reachability of the weak reference object nor a reference in the finaliser affects the reachability of the key or value.

For weak references the above definition of reachability can be extended in the following way. An object is reachable if at least one of the following statements holds true [18]:

- It is in the *root set*.
- It is directly referenced by another reachable object, apart from a weak reference.

[1] Note that this statement says "if" and not "if and only if".

- It is a weak reference object whose key is reachable.
- It is a value or a finaliser of a weak reference whose key is reachable.

2.2.1. Semi-Explicit Parallelism

While completely implicit parallelism is not profitable enough yet, Trinder et al. introduced a runtime supported semi-explicit parallelism interface for Haskell with *Glasgow parallel Haskell* in 1998 [31]. Based upon this, Marlow et al. presented a more flexible formulation of strategies for semi-explicit parallelism in 2010 [25]. For both interfaces, the runtime system manages most of the parallel execution, requiring the programmer to indicate these expressions that might be usefully evaluated in parallel. Both interfaces describe deterministic parallelism, meaning that the parallelisation does not affect the behaviour of the program.

Basic Combinators

In Haskell parallelism is introduced by the combinator *par*.

$$par :: a \longrightarrow b \longrightarrow b$$

The evaluation of the expression $p \grave{} par \grave{} e$ allows the evaluation of p in parallel and then evaluates e. Often, p itself appears somewhere in the expression e. Thereby, *par* is only strict in its second argument; $\bot \grave{} par \grave{} e$, with $\bot$ being a failure, would thus be evaluated to e. When $p \grave{} par \grave{} e$ is evaluated, we say that p becomes *sparked*. It is added to the *spark pool*, which is a buffer of sparked computations. If a processor becomes idle, it starts to evaluate a spark from the spark pool. However, sparked expressions are not necessarily evaluated in parallel if their evaluation is requested earlier by the main evaluation. *Sparks* have very little overhead compared to the evaluation in a separate Haskell thread. Note that *par* does not affect the semantics of the expression and it can, therefore, be omitted to retrieve a sequential program.

Similar to *par*, the Haskell Prelude contains the function *seq*.

$$seq :: a \longrightarrow b \longrightarrow b$$

$s \grave{} seq \grave{} e$ denotes the evaluation of the first argument s to weak head normal form[2] before returning the result of the second argument e. In contrast to *par*, *seq* is strict in both arguments so that the compiler may rewrite the above expression to $e \grave{} seq \grave{} s \grave{} seq \grave{} e$. Although, this usually is not a problem when expressing strictness, it may be a problem when annotating code for parallelism: we may want to evaluate s before e, because e has already been annotated for evaluation in parallel. For this, we have the function *pseq*.

$$pseq :: a \longrightarrow b \longrightarrow b$$

[2]Weak head normal form means that the head of an expression cannot be evaluated any further.

pseq is only strict in its first argument and therefore $s \grave{} pseq \grave{} e$ may not be rewritten as described above. *pseq* and *par* are both part of the original Glasgow parallel Haskell. This interface has been used in the original parallel search strategy for KiCS2 by Reck and Fischer [30].

Eval Monad

Based on the original strategies from [31], Marlow et al. introduced a new formulation of parallelisation strategies in 2010 [25]. It is based on the type *Strategy*, which is a function that embodies a parallel evaluation strategy.

```
data Eval
type Strategy a = a → Eval a
runEval :: Eval a → a
```

A strategy does an arbitrary amount of evaluation on its argument in parallel or in sequence and it has to return the argument itself. *Eval* is a *monad* and is called the *evaluation order monad*. It is used to define new strategies by combining existing ones. *Eval* is a *strict identity* monad and therefore can be used to control the evaluation order [25, 33].

Existing strategies encompass the following:

r0 performs no evaluation at all.

rseq evaluates its argument to weak head normal form.

rdeepseq fully evaluates its argument.

rpar sparks its argument for evaluation in parallel.

The monadic notation gives us a concise way to express the evaluation order. Consider the different formulation of the same rule of *nfib* as seen in figure 2.2.

```
nfib n =
  let x = nfib (n - 1)
      y = nfib (n - 2)
  in x `par` (y `pseq` x + y + 1)
nfib n = runEval $ do
  x ← rpar (nfib (n - 1))
  y ← rseq (nfib (n - 2))
  return (x + y + 1)
```

Figure 2.2.: The parallel rule of *nfib* written using basic combinators and rewritten using strategies.

We can think of the evaluation monad as a *embedded domain-specific language* (*EDSL*), allowing us to express evaluation order in the language Haskell, which otherwise has no clear evaluation order.

Fizzled Sparks

When an expression in the spark pool is required by the main evaluation during its normal execution, it will be evaluated immediately by this thread. As a result, the spark pool may also contain values instead of unevaluated expressions. We say, this spark is *fizzled*. As it is already evaluated, there is no use in evaluating this spark in parallel. The runtime system removes these fizzled sparks from the spark pool, consequently the garbage collector may remove the values they refer to as long as they are not referenced elsewhere.

Speculative Parallelism

Using the new formulation, the runtime system also supports *speculative parallelism* where an expensive evaluation is parallelised even though its value is not known to be required later [25]. In its implementation, the spark pool contains weak references (section 2.2), so the garbage collector only retains sparks that are otherwise reachable from the root of the program.

2.2.2. Concurrent Haskell

Concurrent Haskell is an extension to Haskell adding support for explicitly threaded concurrent programming [14, 17]. Concurrent Haskell was created to write concurrent applications such as interactive and distributed systems in contrast to parallel applications. In these concurrent applications, concurrency is often used to increase its responsiveness.

The authors of Concurrent Haskell prefer implicitly threaded parallelisation from section 2.2.1 for parallelism which is increasing the performance by exploiting multiprocessors. However, Concurrent Haskell has been proven useful to implement parallelism as well, because it does not have Glasgow parallel Haskell's limitation of being semantically deterministic (see section 4.4).

Programming in Concurrent Haskell is based upon the following two basic concepts.

- Threads and a mechanism to initiate new threads.
- Atomically mutable state for communication between multiple threads.

Based on these foundations, more elaborate concepts have been developed, among which are:

- *Asynchronous exceptions* were developed to cancel foreign threads [24].

- *Software Transactional Memory* (*STM*) allows safe compositions of access on shared state using optimistic synchronisation [11].

All these concepts have been used for this thesis to implement parallel search strategies. Threads and basic communication through mutable variables are used for the implementation *fair search strategies* in section 4.5 and the *bag of tasks* in section 5.2. The implementation of the bag of tasks also uses STM for the communication between multiple worker threads. Asynchronous exceptions are used to cancel foreign threads as described in section 5.1 and for various inter thread communication of the fair search strategy defined in section 4.5.3.

Threads

One basic concept of the explicit concurrency in Concurrent Haskell are *threads*. Threads are sequences of actions that can be executed independently; in Haskell actions are values of type *IO a* and therefore actions that might perform input/output operations. Because threads are independent from each other they may be executed in parallel. A new thread can be started using *forkIO*:

$$forkIO :: IO\ () \rightarrow IO\ ThreadId$$

forkIO takes the action to be performed in parallel as its argument and returns the action starting a new thread and delivering its unique identifier, its *ThreadId*. When *forkIO* is executed, a new thread will be started that runs concurrently with all other threads on the system. If multiple threads have *effects*, the effects will be interleaved.

The interleaving of effects can be illustrated with the example

```
main = forkIO (write 'a') >> write 'b'
write x = putChar x >> write x
```

which results in a random interleaving of as and bs for example in the following output.

```
abbaababababbaaaaaabbaabbbbb...
```

Note that this interleaving is non-deterministic, so we may sometimes get strings with only one letter.

In GHC threads are extremely lightweight; typically a thread requires less than a hundred bytes plus its own stack. The size of the stack is dynamic rather than static, so it can grow and shrink with the demand of the thread. While in theory the number of supported threads is in the millions, practically the number of threads is limited by the available memory, because the size of a thread's stack can grow to significant amounts. This phenomenon leads to problems in the memory consumption of the fair search strategy, see section 6.3.3.

A Haskell system may implement preemptive multitasking or cooperative multitasking; GHC, the Haskell implementation used in this thesis, does preemptive multitasking. Therefore, it occasionally stops the running thread and starts a scheduler to decide which thread to run next. This happens when a thread does memory allocation.

For cooperative multitasking, where the thread itself has to initiate a *context switch*, there is the action *yield*.

$$yield :: IO\ ()$$

In case of a cooperative multitasking environment, *yield* forces another thread to be executed next if there are other runnable threads. In a preemptive multitasking environment the action allows a context switch leaving it to the runtime system to actually decide whether it does a context switch.

Mutable Variables

The most basic communication abstraction in Concurrent Haskell is the mutable variable, an *MVar*. An *MVar* can be thought of a box which may either contain a value or be empty. Its state is shared between all threads and it has the following interface.

$$\begin{aligned}
&\textbf{data}\ MVar\ a \\
&newEmptyMVar :: IO\ (MVar\ a) \\
&putMVar \quad\quad :: MVar\ a \to a \to IO\ () \\
&takeMVar \quad\quad :: MVar\ a \to IO\ a
\end{aligned}$$

The action *newEmptyMVar* creates a new empty *MVar*, *putMVar* puts a value into the given *MVar*, but blocks if it is already filled, and *takeMVar* takes the value out of the *MVar* and blocks if it is empty.

With this interface, it is already a generalisation for various concurrency abstractions:

- A mutable variable of type *MVar* () is a binary semaphore or a lock with the signal and wait operations implemented as *putMVar* () and *takeMVar*.
- An *MVar a* can be seen as an one-place channel to be used for asynchronous communication between threads.
- The mutable variable may contain a state shared between multiple threads, which can be modified by a pair of *takeMVar* and *putMVar*.

Channels

A *channel* is a synchronisation tool which allows one or multiple threads to write values into it and one or multiple threads to read these values in the same order. As seen above, an *MVar* can already be used as a channel with a limited capacity. Additionally, Concurrent Haskell provides an unlimited channel *Chan* with the following interface:

```
data Chan a
newChan  :: IO (Chan a)
writeChan :: Chan a → a → IO ()
readChan  :: Chan a → IO a
```

The action *newChan* creates a new channel, *writeChan* writes a value into the channel and *readChan* reads a value from the channel. *readChan* blocks on an empty channel whereas *writeChan* does not block and always succeeds.

This unlimited channel is in fact implemented using mutable variables. Its implementation is described by Peyton Jones et al. [14] or in more detail by Marlow [17].

For this thesis, we use channels to return values of evaluations from other threads to the main thread. The main thread collects these values.

Asynchronous Exceptions

An important concurrent language feature is *asynchronous exceptions* (Marlow et al. 2001 [24]). Asynchronous exceptions, in contrast to synchronous exceptions, can be thrown by one thread to another. While the use of such exceptions seems to contradict modularity, it is often an useful tool to signal timeouts and to terminate threads which calculate values that are no longer required by the system (section 5.1).

Exceptions in Haskell are values of types which instantiate the type class *Exception*.

```
class (Typeable e, Show e) ⇒ Exception e
```

Haskell exceptions can be thrown both synchronously and asynchronously. Throwing an exception synchronously can be done even in pure code, whereas asynchronously throwing an exception to another thread requires being in the *IO* monad.

```
throw    :: Exception e ⇒ e → a
throwIO  :: Exception e ⇒ e → IO a
throwTo  :: Exception e ⇒ ThreadId → e → IO ()
```

In contrast to the throwing of exceptions, exceptions can only be caught in the *IO* monad.

```
catch :: Exception e ⇒
         IO a            – computation to run
      → (e → IO a)       – exception handler
      → IO a
```

When an exception is thrown – either during the evaluation of pure code or during the execution of an *IO* action – the running thread either stops completely, or, if the exception is thrown inside a catch, the exception handler is run. To catch exceptions in the evaluation of pure code, the module *Exception* also provides a function to force the evaluation of its argument to weak head normal form in the *IO* monad.

```
evaluate :: a → IO a
```

Asynchronous signaling or killing can occur at any point of the target thread's execution, even when a lock is held. The lock will then not be properly released. This makes it necessary to mask the receiving of asynchronous exceptions for certain periods.

```
mask                :: ((forall a ∘ IO a → IO a) → IO b) → IO b
uninterruptibleMask :: ((forall a ∘ IO a → IO a) → IO b) → IO b
```

Masking asynchronous exceptions means that threads, attempting to raise an exception in the masked thread, block until asynchronous exceptions are unmasked again. Masking exceptions with *mask* is interruptible, meaning that exceptions may still be received while the execution is blocked, for instance when blocking on an *MVar*. As its name suggests, *uninterruptibleMask* is not interruptible and therefore guarantees that no asynchronous exception is received in the evaluated code.

The two masking functions *mask* and *uninterruptibleMask* provide a function to restore the previous masking state to the masked code (type *forall a ∘ IO a → IO a*). However, this function does not guarantee that asynchronous exceptions are unmasked again, but rather restores the state prior to the masking. If the previous state was also *masked*, it would simply restore the masked state. The only possibility to unmask all asynchronous exceptions masked in the calling code is to start a new thread with *forkIOWithUnmask*.

```
forkIOWithUnmask :: ((forall a ∘ IO a → IO a) → IO ()) → IO ThreadId
```

Similarly to *mask*, it provides a function to the thread which performs its argument in another mask state. This time, it completely unmasks all asynchronous exceptions. Thus, it is often used in library code which throws asynchronous exceptions itself, for example in the code explained in section 4.5.

Software Transactional Memory

Concurrent programming, even if we only consider concurrency between threads in the same process, is extremely difficult. Using programming techniques based on locks easily results in wrong results, deadlocks, and bad performance. Moreover, lock-based concurrency often conflicts with modularity.

Consider the following Haskell interface for bank accounts where both functions, *dispose* and *withdraw*, are individually correct; they are thread-safe and atomic: no intermediate state is visible to another thread.

$$dispose \;\; :: Account \longrightarrow Amount \longrightarrow IO\ ()$$
$$withdraw :: Account \longrightarrow Amount \longrightarrow IO\ ()$$

However, correctly composing both actions to implement a bank transfer is not possible. Independent of the order of the composition, an intermediate state would be visible to another thread. If the second action blocks, this intermediate state could be even visible for a long time.

More promising is the use of *software transactional memory* (STM) [11], which allows us to atomically compose memory transactions. Using STM, the type signature of the functions introduced above is a little different.

$$dispose \;\; :: Account \longrightarrow Amount \longrightarrow STM\ ()$$
$$withdraw :: Account \longrightarrow Amount \longrightarrow STM\ ()$$

We want to perform actions, such as *dispose* and *withdraw*, atomically.

$$atomically :: STM\ a \longrightarrow IO\ a$$

Like *IO*, *STM* is a monad, so we are able to actually compose both actions. To implement the desired bank transfer we simply compose both actions in an arbitrary order.

$$transfer :: Account \longrightarrow Account \longrightarrow Amount \longrightarrow STM\ ()$$
$$transfer\ from\ to\ amount = withdraw\ from\ amount \gg dispose\ to\ amount$$

Note that an implementation of *withdraw* may block when there is no money on the bank account *from* and perhaps also *dispose* may be blocking when the account has been locked by the bank. In both cases, no action will be performed and the whole transaction will be *retried* once the global state changes. Such a retry can be triggered with the action *retry*.

$$retry :: STM\ ()$$

It does not necessarily trigger the execution being run again directly, but stops the evaluation of the action. When one of the read variables is changed, the transaction will be retried. The variables, which are managed by the software transactional memory system, are called *transactional variable*, *TVar*.

$$\textbf{data}\ TVar\ a$$
$$newTVar \;\; :: a \longrightarrow STM\ (TVar\ a)$$
$$readTVar \;\; :: TVar\ a \longrightarrow STM\ a$$
$$writeTVar :: TVar\ a \longrightarrow a \longrightarrow STM\ ()$$

Knowing how to interact with the transactional memory, we are now able to define the *withdraw* function declared above. It can be seen in figure 2.3.

Note that the software transactional memory system has to restart transactions in two cases:

- A transaction reaches the *retry* command.

```
type Amount = Int
data Account = Account (TVar Amount)
withdraw (Account balanceVar) amount = do
  balance ← readTVar balanceVar
  if balance < amount
    then retry
    else writeTVar balanceVar (balance − amount)
```

Figure 2.3.: Implementation of a bank account's *withdraw* action using software transactional memory.

- Another transaction changes one of the read values and therefore makes the transaction invalid.

Because transactions might get aborted, they must not do anything irrevocable; in particular, *IO* actions are not allowed.

Using transactional variables, it is possible to implement more sophisticated types like transactional channels (*TChan*) and transactional mutable variables (*TMVar*), which are also provided in the Haskell package *stm*.

2.2.3. Runtime System

GHC's runtime system supports thousands of Haskell threads and millions of sparks by multiplexing them onto a handful of system threads. These system threads are called *capabilities*. The number of these is set at the start of the runtime system, but it may be increased later. Benchmarks show that it is best to have roughly one capability for each physical CPU core. A capability can either execute a Haskell thread, or, if there are no Haskell threads ready to run, evaluate a spark (see section 2.2.1). More information regarding the multiprocessor support in Haskell is given by Marlow et al. [22].

2.3. KiCS2

KiCS2 compiles Curry, described in section 2.1, to Haskell. The non-determinism in KiCS2 is represented explicitly in the data structures [4]. These data structures are explained in section 2.3.1. These are then translated into search trees explained in section 2.3.2.

2.3.1. Representing Non-Determinism in Data Structures

In a non-deterministic language, an expression can yield multiple values or even no value. Multiple values are represented by an additional *Choice* constructor for each type, for instance

for the type *Bool*:

$$\textbf{data}\ Bool = True \mid False \mid Choice\ Bool\ Bool$$

In section 2.1.5 we introduced the operation *aBool*, which yields either *True* or *False*. It can now be written as:

$$aBool = Choice\ True\ False$$

Because all data types have an additional constructor, all operations based on pattern matching have to be extended so they will not fail on the occurrence of a choice constructor, but move the choice constructor one level higher. We define the boolean negation $\neg$ in Curry like in the following example:

$$\begin{aligned}\neg\ False &= True\\ \neg\ True &= False\end{aligned}$$

The extension to support the choice constructor then moves the constructor one level higher:

$$\begin{aligned}\neg\ False &= True\\ \neg\ True &= False\\ \neg\ (Choice\ x\ y) &= Choice\ (\neg\ x)\ (\neg\ y)\end{aligned}$$

While this allows us to introduce non-determinism, it does not reflect Curry's *call-time choice semantics* (see section 2.1.5) correctly. Consider the operation *xorSelf*, which we used to explain the call-time choice semantics.

$$\begin{aligned}&xor\ True\ &&True\ &&= False\\ &xor\ True\ &&False\ &&= True\\ &xor\ True\ &&(Choice\ x\ y)\ &&= Choice\ (xor\ True\ x)\ (xor\ False\ y)\\ &xor\ False\ &&y\ &&= y\\ &xor\ (Choice\ x\ y)\ z\ &&\ &&= Choice\ (xor\ x\ z)\ (xor\ y\ z)\\ &xorSelf\ x = xor\ x\ x\end{aligned}$$

As *xorSelf* does not depend on pattern-matching, there is no need to transform it in any way. Let us again have a look at the expression *xorSelf aBool* and its evaluation.

$$\begin{aligned}xorSelf\ aBool &\rightarrow xorSelf\ (Choice\ True\ False)\\ &\rightarrow xor\ (Choice\ True\ False)\ (Choice\ True\ False)\\ &\rightarrow Choice\ (xor\ True\ (Choice\ True\ False))\ (xor\ False\ (Choice\ True\ False))\\ &\rightarrow Choice\ (Choice\ (xor\ True\ True)\ (xor\ True\ False))\ (Choice\ True\ False)\\ &\rightarrow Choice\ (Choice\ False\ True)\ (Choice\ True\ False)\end{aligned}$$

The choices in this result represent different possible values. If we want to show all values of an expression, we have to enumerate all values contained in the choices. In this case, these are *False*, *True*, *True*, and *False*. Indeed, these are exactly the results we would get when interpreting the Curry program as a term rewriting system, but it does not reflect the call-time choice semantics. Call-time choice semantics does not allow the value *True* for this expression,

because the values of a non-deterministic expression are determined at the time of the function application. To ensure this, different choice instances are uniquely identified. Therefore, every choice-constructor gets an additional identification parameter.

data *Bool* = *True* | *False* | *Choice ID Bool Bool*

The *ID* could, for example, be an integer.

type *ID* = *Integer*

With this additional identifier, the expression *xorSelf aBool* evaluates in the following way:

xorSelf aBool

$\longrightarrow$ *xor* (*Choice* 1 *True False*) (*Choice* 1 *True False*)
$\longrightarrow$ *Choice* 1 (*xor True* (*Choice* 1 *True False*)) (*xor False* (*Choice* 1 *True False*))
$\longrightarrow$ *Choice* 1 (*Choice* 1 *(xor True True*) (*xor True False*)) (*Choice* 1 *True False*)
$\longrightarrow$ *Choice* 1 (*Choice* 1 *False True*) (*Choice* 1 *True False*)

To get call-time choice semantics here, we have to make *consistent selections*: we have to select the same branch for all choice constructors with the same identifier. In this case, selecting the left branch in the outer choice means also selecting the left branch in the inner choice resulting in the desired value *False*. Selecting the right branch in the outer choice means selecting the right branch in the inner choice, also resulting in the value *False*.

This implementation requires the creation of fresh identifiers during the computation, which is a non-trivial problem in lazy functional languages. In fact, in KiCS2 it is possible to select from multiple implementations of the identifier supply.

In contrast to evaluations yielding multiple results, there are also those that *fail*: those do not yield any result. A failure does not result in an abortion of the whole computation, but can be considered a part of the computation which does not produce a result. In KiCS2, a failure is represented as an additional constructor of each data type called *Fail*.

data *Either a b* = *Left a* | *Right b* | *Choice ID* (*Either a b*) (*Either a b*) | *Fail*

The implementation of the function *left*, in Curry

```
left :: Either a b → a
left (Left a) = a
```

can be translated to Haskell using an additional rule matching any values when all other rules failed.

```
left (Left x)         = x
left (Choice i x y) = Choice i (left x) (left y)
left _                = Fail
```

Note that *left* results in a failure if the argument is *Right a*, or if it already is a failed computation. Thus, it propagates the failed state of its argument.

2.3.2. Search Tree

The operations generated in section 2.3.1 create structures containing choices, failures, and finally values. To extract the value of a computation, we have to enumerate all values in the choice tree in some order, either sequentially or in parallel. To provide a common interface for these enumerations, we provide a simplified data structure of a search-tree.

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

The search tree is very similar to the data structure above. It can either be a failure (*None*), one value (*One*), or a choice (*Choice*). Each occurence of one of these constructors is called *node*. Both, *None* and *One* constructors are the *leaves* of the search tree.

In contrast to the structure above, forbidden branches – those with different decisions for two choices with the same *ID* – are already eliminated. Thus, the *ID* is not necessary in the choice constructor. The function *try* to compute this search tree is omitted here, but the implementation is discussed in [4].

try :: *a* → *SearchTree a*

Note that in the search tree created by *try*, each expression in a *One* constructor is already evaluated to *normal form*. Normal form means that there is no function application left in the expression. If it had a function application left, it could introduce a choice operator, ?, and therefore we would not have just one but possibly multiple results.

Those search trees can very well be *infinite*, like the search tree of the computation sketched in figure 2.4.

main = *xs* ++ [*x*] **where** *x*, *xs* **free**

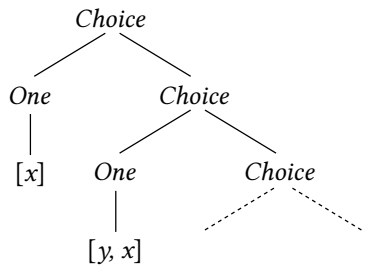


Figure 2.4.: An infinite search tree.

Representing non-deterministic results in a data-structure rather than as a computation allows us to define different strategies to explore the search space. In fact, KiCS2 allows the user to select a search strategy. It initially comes with depth-first search (section 4.2.1), breadth-first search (section 4.2.2), iterative deepening, and a simple parallelised depth-first strategy (section 4.3.1). Additionally, the user is able to choose between different forms of displaying the values of a computation. The user may choose either to print all solutions, to print only one solution, or to print one solution after the other by the user's requests.

3. Evaluation Criteria

To design search strategies, it is necessary to keep the criteria in mind, which are used in the evaluation. This chapter gives a short introduction to those criteria. The evaluation of the search strategies is described in chapter 6.

3.1. Resource Consumption

Speeding up the computation of the search results by exploiting multiple processors is a main goal of this thesis. Thus, resource consumption is the most obvious criterion. Apart from the needed computation time, we also have to keep in mind the memory consumption of the search strategies.

3.2. Completeness

A huge problem in programming are calculations that diverge and therefore do not terminate. In Curry, different search strategies may result in different termination behaviours when being applied to the same program. Programs exposing this behaviour create search trees, some branches of which converge whereas others diverge. We distinguish between various possible sources of divergence.

We call a search strategy *complete* if the strategy finds all available values in all search trees. It is complete with regard to a certain category of search trees if it finds all available values in search trees of this category. The following text presents categories, which are important for this thesis.

Finite Trees

The most simple category of search trees are *finite trees*. Because a finite tree is a finite data structure, it can be evaluated to normal form. The enumeration of all values in this search tree is trivial and therefore all presented search strategies are complete regarding finite trees.

Search Trees with Infinite Non-Deterministic Computations

The following example shows that search trees do not have to be finite, but instead may be infinite:

$$
\begin{aligned}
&[\,] \mathbin{+\!\!+} ys = x \\
&(x : xs) \mathbin{+\!\!+} ys = x : (x \mathbin{+\!\!+} ys) \\
&\mathit{ending}\ x = xs \mathbin{+\!\!+} [x]\ \textbf{where}\ xs\ \textbf{free}
\end{aligned}
$$

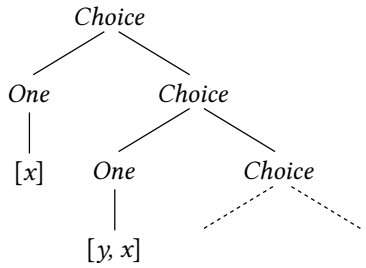


Figure 3.1.: A search tree with an infinite number of choices.

Using this definition, the expression *ending* 1 would be an arbitrary list ending with the value 1. The search tree of this expression can be seen in figure 3.1. As there is an infinite number of lists ending with the value 1, the search tree of this expression has an infinite number of values and also an infinite number of choices. Evaluating the whole search tree to normal form would not terminate as its structure is infinite. Because each level in the search tree has only an finite number of nodes, it is possible to enumerate all values of the search tree level-wise with breadth-first search (section 4.2.2).

Search Trees with Infinite Deterministic Computations

In addition to search trees with infinite numbers of choice constructors, there are also search trees in which the computation of a single tree node, for instance a *One* constructor, does not terminate. Consider the following minimal example[1].

$$
\begin{aligned}
&\mathit{loop} = \mathit{loop} \\
&\mathit{main} = \mathit{loop}\ ?\ ()
\end{aligned}
$$

The search tree for the *main* operation can be seen in figure 3.2. While the result in the left branch cannot be evaluated to normal form, the value on the right, (), is already in normal form. A complete search strategy would have to find the value of () at some point in the calculation.

[1] In KiCS2, a simple *loop* function would be detected to be non-terminating by GHC's runtime system.

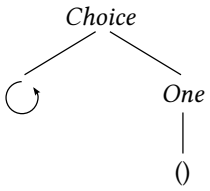


Figure 3.2.: A search tree with a deterministic loop.

Note that the figure does not show the *One* constructor in the left branch even though *loop* is a deterministic computation. The runtime system of KiCS2 does not detect that the calculation of loop is deterministic and so it does not know if, at some point, a non-deterministic choice appears in the calculation. As a result it does not know that there is only one value in the looping branch.

Often, such infinite deterministic computations create data constructors. One possible computation is the evaluation of an infinite data structure.

$$ones = 1 : ones$$
$$main = ones\ ?\ [\,]$$

Again the value on the right, [ ], is already in normal form and has to be found by any complete search strategy. However, most strategies would run into an infinite computation during the evaluation of *ones*.

3.3. Stopping the Computation

Often, a programmer only wants to know one value of an expression or just wants to know about the existence of a value. We already saw such an example with the function *hasChild* (section 2.1.6), which returns exactly *one success* if there the value provided as the argument has at least one child.

$$hasChild\ x \mid \neg\ (isEmpty\ ((set1\ child)\ x)) = success$$

When we use *hasChild* in our code, we don't want the computation to continue unnecessarily after computing the first value.

While this is relatively easy for sequential search strategies, it is not trivial for parallel search strategies. With a parallel search strategy, we may have started multiple threads to calculate the values of the expression, but we are not interested in those anymore. Starting a parallel computation without knowing that we will need its result in the end is called *speculative parallelisation*. All those parallel computations have to be stopped once the first result is available.

4. Search Strategies

Having the possibility to choose between multiple search strategies is an important feature of KiCS2. The system can easily be extended with additional strategies such as the parallel strategies we introduce in this thesis. The next sections describe how these search strategies are integrated into the complete system (section 4.1), the implementation of the default sequential strategies (section 4.2), and the idea and implementation behind the newly developed parallel search strategies. These include deterministic parallel search strategies (section 4.3), non-deterministic search strategies using a bag of tasks approach (section 4.4), and fair search strategies (section 4.5).

4.1. Search Strategies as Part of the KiCS2 System

This section describes how sequential and parallel search strategies are integrated with the rest of KiCS2. Figure 4.1 shows the dependencies between the modules containing the most important parts of the implementation of non-deterministic search. The search strategies provided as part of the runtime system are defined in the module *Search*. In addition to these built-in search strategies there are also some search strategies defined in Curry itself as part of the standard Curry library in the module *SearchTree*. *SetFunctions* are based upon this and thus use the same search strategies.

KiCS2 links the *main program* against *Search* to use one of the search strategies for its main goal. Furthermore, the *main program* can make use of the modules *SetFunctions* and *SearchTree* for encapsulated search. Though, it is better to use *SetFunctions*, because using *SearchTree* directly is deprecated.

As part of this thesis, we added parallel search strategies to the module *Strategies* in KiCS2's runtime system. These have been made available to the user in two ways: it is possible to use these strategies as a search strategy for the main goal and there is a Curry interface to these search strategies in *ParallelSearch*. In contrast to *SetFunctions*, the interface in *ParallelSearch* can only be used in the *IO* monad. Thus, certain profitable search strategies have been reimplemented in *External_SearchTree*, the Haskell implementation for parts of *SearchTree*, as well.

The parallel strategies are implemented as search strategies on a search tree data type called *SearchTree*.

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

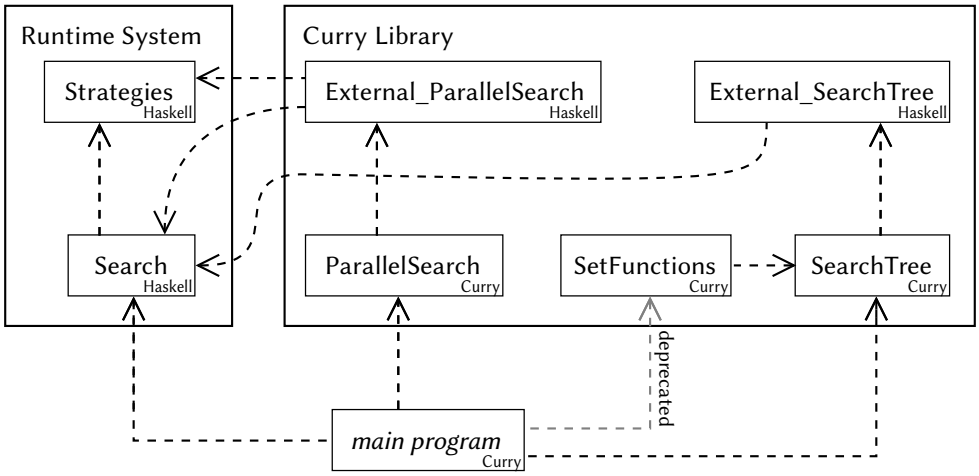


Figure 4.1.: Dependencies between relevant modules of the KiCS2 with Parallel Search.

All parallel search strategies are defined as a function transforming this search tree into a list. Some of these are defined in terms of *IO* actions and others are defined as pure code.

$$search :: SearchTree\ a \to [a]$$
$$searchIO :: SearchTree\ a \to IO\ [a]$$

When the search is defined as *IO* actions, the evaluation of the returned list happens lazily.

All strategies depend on the search tree being generated by *searchMSearch* or by *encapsulatedSearch*. *searchMSearch* is for the top-level search and *encapsulatedSearch* is for the encapsulated search; *encapsulatedSearch* depends on *searchMSearch*, itself. The Curry bindings in *ParallelSearch* directly translate the results to Curry for the functions described in section 7.

For each top-level search strategy, there is a function to transform the non-deterministic expression into a *monadic list*.

$$compute :: NormalForm\ a \Rightarrow NonDetExpr\ a \to IO\ (List\ IO\ a)$$

A monadic list is a list containing nested monadic actions and is defined in the following way.

```
data List m a
  = Nil                      – Empty list
  | Cons a (m (List m a))    – List constructor
```

It is used to delay *IO* actions until their results are requested explicitly. This way, it is possible to offer different output variants like printing all values, printing just the first value, or printing the values on the user's demand.

4.2. Sequential Search Strategies

To traverse the search tree (section 2.3.2) and convert it into a sequence, we define different search strategies. KiCS2 initially provides a set of sequential search strategies, including depth-first search, breadth-first search, and iterative deepening. Two of these, namely depth-first search and breadth-first search, have proven to be an useful base for parallel search strategies and therefore they are described in the following sections.

4.2.1. Depth-First Search

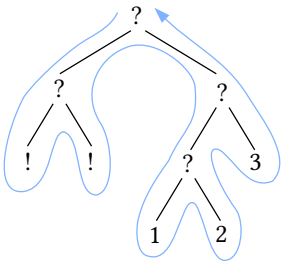


Figure 4.2.: A depth-first search over a search tree.

The *depth-first strategy* is similar to the evaluation strategy of implementations using a backtracking approach in the logic language Prolog. It starts traversing the search tree at the root node, the topmost node in figure 4.2, and then first traverses the left subtree before visiting the right subtree. Both subtrees are then traversed with the same strategy. As a result the search follows one branch down to its leaf before vising the next branch.

We define all search strategies as strategies to traverse the tree defined in section 2.3.2. The constructor *None* represents a failure, *One x* represents a single value, and *Choice l r* represents a non-deterministic choice between its two subtrees. Therefore, the search strategy has the following type signature:

$$dfsSearch :: SearchTree\ a \to [a]$$

The simple constructors *None* and *One* can be handled easily as those search trees have only one possible list representation: the empty list and the list containing just one element.

$$\begin{aligned} dfsSearch\ None &= [\,] \\ dfsSearch\ (One\ x) &= [x] \end{aligned}$$

At a non-deterministic choice, the results of the right subtree are appended to the results of the left subtree. When the list is evaluated, the strategy first searches the left subtree and then continues with the right subtree.

$$dfsSearch\ (Choice\ l\ r) = dfsSearch\ l +\!\!+ dfsSearch\ r$$

As this search follows each branch down to its leaf, it may also follow an infinite branch before returning an existing leaf in the search tree. Thus, the depth-first strategy is only complete for finite search trees.

4.2.2. Breadth-First Search

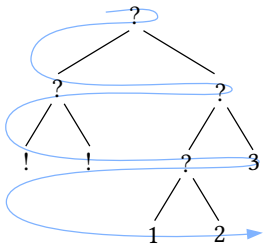


Figure 4.3.: A breadth-first search through a search tree.

Breadth-first search partly solves the completeness problem of the depth-first search. This strategy inspects the search tree level by level and thus does not run into infinite choice structures before evaluating each leaf. The order in which the nodes are visited can be seen in figure 4.3.

To inspect the search tree level by level, the nodes of the current level are stored in a list. In the first level there is only the root node.

$$bfsSearch :: SearchTree\ a \longrightarrow [a]$$
$$bfsSearch\ t = bfs\ [t]$$

The auxiliary function *bfs* traverses the list on from the current level.

$$bfs :: [SearchTree\ a] \longrightarrow [a]$$
$$bfs\ [\,] = [\,]$$
$$bfs\ ts = values\ ts +\!\!+ bfs\ (children\ ts)$$

To define the function *bfs* we need two additional auxiliary functions. *values* finds all *One* constructors in the current level and returns their values. The other function, *children*, looks for all occurrences of a *Choice* constructor in the current level and returns the corresponding subtrees. The result list then consists of the values in the first level concatenated with the list of the values from the levels below. Both auxiliary functions can be seen in figure 4.4.

```
values :: [SearchTree a] → [a]
values []             = []
values (One x : ts) = x : values ts
values (_     : ts) = values ts

children :: [SearchTree a] → [SearchTree a]
children []                 = []
children (Choice x y : ts) = x : y : children ts
children (_          : ts) = children ts
```

Figure 4.4.: Auxiliary functions to implement breadth-first search.

4.3. Order-Preserving Parallel Search Strategies

Haskell's parallelisation libraries (section 2.2.1) have been developed for deterministic parallelism. Thus, the parallel evaluation order has no influence on the results. Search strategies defined using these libraries return the same result lists as an equivalent sequential strategy. The order of the elements in the result lists is deterministic and will not change between multiple runs of the program.

The semi-explicit parallelism libraries are easy to use and implemented with a very small overhead by Glasgow parallel Haskell compared to manual parallelisation with threads.

Based on these libraries, there is an original parallel strategy available with KiCS2, implemented using Haskell's simple *par* construct (section 4.3.1). In this thesis, this strategy is ported to use the new strategies for semi-explicit parallelism (section 4.3.2). Based on this strategy, we try various possible enhancements to decrease the runtime.

4.3.1. Original Approach to Deterministic Parallel Depth-First Search

This section explains the original parallel search strategy presented by Reck and Fischer [30] in 2009. It is similar to the depth-first search strategy introduced in section 4.2.1. This strategy is called *parSearch*.

```
parSearch :: SearchTree a → [a]
parSearch None       = []
parSearch (One x)    = [x]
parSearch (Choice l r) =
  let rs = parSearch r
  in  rs `par` (parSearch l ++ rs)
```

Figure 4.5.: Original parallel search strategy of KiCS2 (*parSearch*).

The implementation can be seen in figure 4.5. The parallelisation happens in the equation for the *Choice* constructor. If a *Choice* constructor appears in the search tree, the evaluation of its right subtree is *sparked* for evaluation in parallel. During the evaluation of the first element of the result list, all right branches are sparked. Once a thread becomes idle, it may take a spark from the spark pool and evaluate its result list to weak head normal form. This, again, creates sparks for all right branches to be evaluated in parallel.

The main thread still evaluates the tree in the same order as depth-first search. If another thread already evaluated the corresponding spark, it finds already evaluated right branches in the search tree. As the ordering in the resulting list is unchanged compared to sequential depth-first search, this strategy also follows infinite branches.

4.3.2. Deterministic Parallel Depth-First Search with Strategies

Since the strategies in the *Eval* monad provide a higher level interface for parallelism with sparks (see section 2.2.1), we rewrite the above strategy with this monad. This makes it more convenient to implement more sophisticated strategies based on the original one.

Direct Translation

The direct translation of the above strategy is called *splitAll1* as it splits the evaluation at each choice in one spark and one part being evaluated by the main computation. In comparison to *parSearch* only the equation for the *Choice* constructor is different, as it is the only equation introducing parallelism.

```
splitAll1 (Choice l r) = runEval $ do
  rs ← rpar (splitAll1 r)
  ls ← rseq (splitAll1 l)
  return (ls ++ rs)
```

The code first sparks the evaluation of the right subtree r for its evaluation in parallel before evaluating the left subtree l sequentially to weak head normal form. The evaluation of l may itself create new sparks if l contains *Choice* constructors. In contrast to the implementation of *parSearch*, the strategy *rseq* enforces the evaluation of *ls* before appending *rs* to *ls*. However, the implementation of ++ (append) would also immediately force the evaluation of *ls*, so it should not make any difference for the evaluation time.

Separation of Evaluation and Traversing

The use of the evaluation monad encourages defining the evaluation of a data-structure separately from its definition and processing. An evaluation strategy is then expressed with a value of the type *Strategy*.

```
type Strategy a = a → Eval a
```

Above, we always define an evaluation "strategy" of type $a \longrightarrow Eval\ b$, or more specific $SearchTree\ a \longrightarrow Eval\ [a]$.

The following strategy to evaluate a search tree in parallel is comparable to the evaluation order of the previously defined search strategy *splitAll*1. In contrast to *splitAll*1 it does not force the sequential evaluation of the left subtree.

```
parTree :: SearchTree a → Eval (SearchTree a)
parTree (Choice l r) = do
  r2 ← (rpar `dot` parTree) r
  l2 ← (rpar `dot` parTree) l
  return (Choice l2 r2)
parTree t = r0 t
```

Based on this evaluation strategy, it is possible to write an alternative implementation of *splitAll*1. This is called *splitAll*2 and is simply defined by the normal, "sequential", depth-first search applied to the search tree being evaluated using the strategy *parTree*.

```
splitAll2 t = dfsSearch (t `using` parTree)
```

Additionally, it is possible to define other search strategies based on *parTree*, like a breadth-first search (section 4.3.4).

Sparking the Evaluation of the Whole List

Let us first consider the definition of *splitAll*1 again.

```
splitAll1 (Choice l r) = runEval $ do
  rs ← rpar (splitAll1 r)
  ls ← rseq (splitAll1 l)
  return (ls ++ rs)
```

rpar sparks the evaluation of its argument to weak head normal form. As this evaluation is also done using *splitAll*1, it also sparks the evaluation of all right subtrees. The spark itself does not evaluate the whole right subtree but only the head of the list. Though, the remainder of the list may be evaluated by the sparks created here. The evaluation of the spark does not wait for the result of this computation. In contrast to this, we define *splitAll*3, in which all sparks evaluate all results of the right subtree. To evaluate all values in a list, there is a function that transforms a strategy on a list element into a strategy on the whole list. This strategy applies the given strategy to each list element.

```
evalList :: Strategy a → Strategy [a]
```

Using *evalList*, we can define a strategy evaluating each element of our results to weak head normal form.

```
evalList rseq (splitAll3 r)
```

This would evaluate each of the elements of the resulting list in sequence. As we want to evaluate the list in parallel to the left subtree, we use *rparWith* which transforms a strategy for evaluation in sequence into one which evaluates its argument in a spark.

$$rparWith :: Strategy\ a \longrightarrow Strategy\ a$$

Therefore, the combination of both, *rparWith* (*evalList rseq*), sparks the sequential evaluation of its argument. Thus, the parallelising rule of *splitAll*3 is defined as follows:

```
splitAll3 (Choice l r) = runEval $ do
  rs ← rparWith (evalList rseq) (splitAll3 r)
  ls ← rseq (splitAll3 l)
  return (ls ++ rs)
```

4.3.3. Reducing the Number of Sparks for Parallel Depth-First Search

Depending on the structure of the search tree, the evaluation with one of the strategies presented so far may result in many sparks being created. Even though the implementation of sparks is very lightweight, reducing the number of sparks and therefore increasing the amount of work per spark may be worthwhile. In this section, we describe various techniques to reduce the number of sparks in depth-first strategies.

Limit the Splitting Depth

The first approach to reduce the number of sparks is limiting the depth to which the tree is evaluated in parallel with a certain threshold [3]. The subtrees below are evaluated with the sequential depth-first search. Figure 4.6 shows a search tree with the subtrees, which would be evaluated sequentially when using a depth limit of 2.

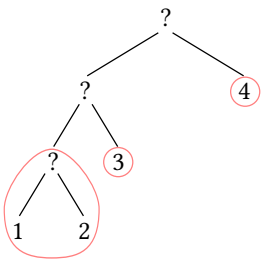


Figure 4.6.: Limiting the depth of parallelisation for parallel depth-first search with a limit of 2.

This strategy is implemented in the function *splitLimitDepth*, which takes the depth of parallelisation as an argument in addition to the search tree. Again, the behaviour for the *One* and *None* constructors remains unchanged.

```
splitLimitDepth :: Int → SearchTree a → [a]
splitLimitDepth _ None     = [ ]
splitLimitDepth _ (One x) = [x]
splitLimitDepth 0 c@(Choice _ _) = dfsSearch c
splitLimitDepth i (Choice l r)       = runEval $ do
  rs ← rpar (splitLimitDepth (i – 1) r)
  ls ← rseq (splitLimitDepth (i – 1) l)
  return (ls ++ rs)
```

If the depth limit is zero, we can continue with the sequential depth-first strategy; if the depth limit is not zero, we do basically the same as for the strategy *splitAll*1, but we decrease the depth limit by one.

Similar to *splitAll*1, *splitLimitDepth* sparks the evaluation of all right subtrees down to the depth limit. At $i \equiv 1$, *rpar* only sparks the evaluation of the first element from the right subtree as it only evaluates the list to weak head normal form. Further elements would have to be requested from the main thread later. Instead of using *rpar*, all elements have to be evaluated similar to *splitAll*3.

```
splitLimitDepth i (Choice l r) = runEval $ do
  rs ← rparWith (evalList rseq) (splitLimitDepth (i – 1) r)
  ls ← rseq                     (splitLimitDepth (i – 1) l)
  return (ls ++ rs)
```

This strategy is most interesting for well-balanced trees, where all subtrees, evaluated with the sequential search strategy, need a similar evaluation effort. On the other hand, if the tree is not balanced, the work will be distributed unevenly to the workers, resulting in low overall parallelisation.

Split Only Right/Left Branches

Having a look at typical search trees shows that most of them are not well-balanced (see the explanation of the programs used for benchmarking in section 6.3.1). Often search trees are very similar to degenerate trees with one very long branch with numerous comparingly small subtrees at one side. Such a nearly degenerate tree can be seen in figure 4.7. As these subtrees are often similarly sized, it may be a profitable choice to create one spark per subtree.

The strategy *splitRight* is optimised for search trees like the one in figure 4.7 and therefore splits the evaluation only in the right subtrees, but evaluates all left subtrees in sequence. For any other aspects, this strategy is similar to *splitAll*3.

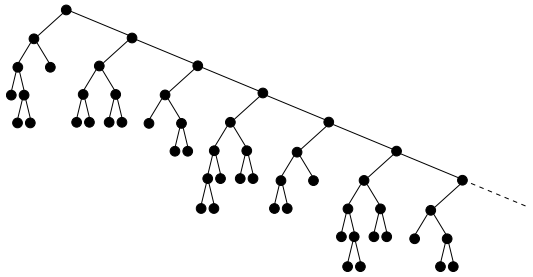


Figure 4.7.: An unbalanced tree.

```
splitRight :: SearchTree a → [a]
splitRight None        = []
splitRight (One x)     = [x]
splitRight (Choice l r) = runEval $ do
  rs ← rparWith (evalList rseq) (splitRight r)
  ls ← rseq                     (dfsSearch l)
  return (ls ++ rs)
```

Note that *splitRight* always creates only one spark. The next spark is then created during the evaluation of this spark, resulting in a constant number of unprocessed sparks in the spark pool.

However, this strategy might result in parallelisation that is too coarse. Sometimes there may be not enough sparks for all threads to work on, especially at the end of the computation. This would reduce the amount of work being done in parallel. Combining this approach with the approach of *splitLimitDepth* fixes this issue by searching fully parallelly at the top of the search tree and only parallelise computations in the right branches of the bottom part of the search tree. Again, we introduce an integer parameter for the depth, which results in the following code:

```
splitRight :: Int → SearchTree a → [a]
splitRight _ None    = []
splitRight _ (One x) = [x]
splitRight 0 (Choice l r) = runEval $ do
  rs ← rparWith (evalList rseq) (splitRight 0 r)
  ls ← rseq (dfsSearch l)
  return (ls ++ rs)
splitRight n (Choice l r) = runEval $ do
  rs ← rparWith (evalList rseq) (splitRight (n - 1) r)
  ls ← rseq (splitRight (n - 1) l)
  return (ls ++ rs)
```

In addition to *splitRight*, there is also the strategy *splitLeft* which mirrors *splitRight*. Note

that *splitLeft* also reverses the order of elements in the result list.

Split Alternating

Even if we have no knowledge about the structure of the search tree, there is a technique to reduce the number of sparks by a linear factor. We create a new spark for every second/third/... choice constructor. A visualisation of this technique can be seen in figure 4.8. A 'P' in the figure means that this part of the search tree may be evaluated in *parallel* and thus being sparked. An 'S' means that this part of the search tree is evaluated sequentially.

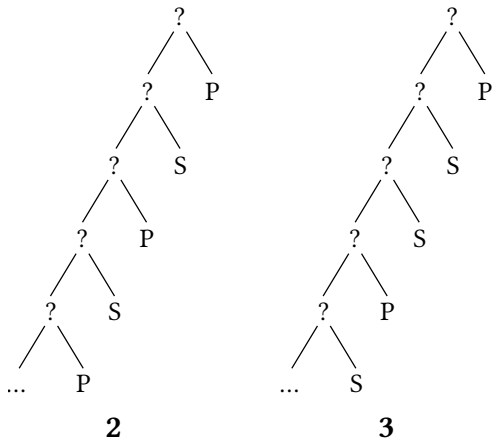


Figure 4.8.: Alternating parallelisation: at every second/third a spark is created.

The search strategy implementing this behaviour is called *splitAlternating*. It takes an additional integer argument and therefore has the following type:

$$splitAlternating :: Int \longrightarrow SearchTree\ a \longrightarrow [a]$$

The additional argument is the distance between two choices that will be sparked for evaluation in parallel. A value of 1 means sparking a subtree at every choice, which makes this strategy equivalent to *splitAll*3.

To implement *splitAlternating*, we need an additional argument to count the distance to the next node to split at. If its value is 1, we split immediately like we did in *splitAll* and reset the counter to *n*. The implementation of *splitAlternating* can be seen in figure 4.9.

Splitting with Exponential Distances

While *splitAlternating* only reduces the number of sparks linearly, it may also be interesting to reduce the number of created sparks even more. To reach this goal, we chose a strategy

```
splitAlternating n = splitAlternating' 1
  where
    splitAlternating' :: Int → SearchTree a → [a]
    splitAlternating' _ None         = []
    splitAlternating' _ (One x)      = [x]
    splitAlternating' 1 (Choice l r) = runEval $ do
      rs ← rparWith (evalList rseq) (splitAlternating' n r)
      ls ← rseq                     (splitAlternating' n l)
      return (ls ++ rs)
    splitAlternating' i (Choice l r) =
      let ls = splitAlternating' (i - 1) l
          rs = splitAlternating' (i - 1) r
      in ls ++ rs
```

Figure 4.9.: Implementation of *splitAlternating*.

similar to *splitAlternating*, but with increasing distances between two choice constructors to create a new spark. For the strategy *splitExponential* the distance between two sparks starts with two and is doubled in each step. The implementation of this strategy is similar to that of *splitAlternating* and it is visible in figure 4.10.

```
splitExponential :: SearchTree a → [a]
splitExponential = splitExponential' 1 2
  where
    splitExponential' _ _ None = []
    splitExponential' _ _ (One x) = [x]
    splitExponential' 1 n (Choice l r) = runEval $ do
      rs ← rpar (splitExponential' n (n * 2) r)
      ls ← rseq (splitExponential' n (n * 2) l)
      return (ls ++ rs)
    splitExponential' i n (Choice l r) =
      let ls = splitExponential' (i - 1) n l
          rs = splitExponential' (i - 1) n r
      in ls ++ rs
```

Figure 4.10.: Implementation of *splitExponential*.

4.3.4. Deterministic Parallel Breadth-First Search with Strategies

While depth-first search is often faster and less memory consuming than breadth-first search, breadth-first search is more complete (see section 4.2.2) and thus it may be profitable to parallelise the evaluation with breadth-first search as well. This section presents different deterministic strategies based on the parallel strategies in the *Eval* monad. All these strate-

gies share the order and the completeness characteristics with the sequential breadth-first search.

Naïve Implementation of Parallel Breadth-First Search

For each step of breadth-first search, the constructors of the current level have to be evaluated to weak head normal form to distinguish between the *Choice*, *One*, and *None* constructors. To evaluate elements of a list in parallel, there is the function *parList*.

```
parList :: Strategy a → Strategy [a]
```

parList creates a strategy on a list which applies the given strategy on each argument in parallel. Therefore, *parList rseq* evaluates all elements of the list to weak head normal form.

```
bfsParallel1 :: SearchTree a → [a]
bfsParallel1 t = bfs [t]
  where
    bfs :: [SearchTree a] → [a]
    bfs [] = []
    bfs (x : xs) = runEval $ do
      rs ← parList rseq xs
      r  ← rseq x
      let rss = r : rs
      return (values rss ++ (bfs (children rss)))
```

Figure 4.11.: The first approach to the parallel breadth-first search.

For the parallel breadth-first search we thereby spark the evaluation of the tail of the list while evaluating its head in sequence. The code of the search strategy can be seen in figure 4.11. After this evaluation we continue the search in the same way as we did for the sequential breadth-first search.

In comparison to the depth-first strategies, the evaluation of the generated sparks is very inexpensive for *bfsParallel*1; here, it is only the evaluation of the first constructor of its subtree. Depending on the degree of non-determinism in a program, these may be very small portions.

Separate Evaluation and Traversion

Similar to the implementation of *splitAll*2 from section 4.3.2, it is possible to define the evaluation of the tree structure separately from the traversal of the tree. To define this strategy we can reuse the implementation of *bfsSearch* presented in section 4.2.2 and the parallel evaluation *parTree* from the definition of *splitAll*2. The most important part of the definition of *parTree* is the rule for the choice constructor. It sparks both subtrees for

evaluation in parallel and then returns the results of these evaluations, which do not have to be evaluated.

```
parTree (Choice l r) = do
  r2 ← (rpar `dot` parTree) r
  l2 ← (rpar `dot` parTree) l
  return (Choice l2 r2)
```

Using this parallel evaluation strategy for a search tree, we can define a breadth-first search analogous to *splitAll*2.

```
bfsParallel2 :: SearchTree a → [a]
bfsParallel2 t =
  bfsSearch (t `using` parTree)
```

Breadth-First Search Using Depth-First Parallel Evaluation

Another option for implementing a parallel breadth-first search strategy is to do a normal breadth-first strategy in the main thread after sparking the evaluation of the tree with depth-first search for the evaluation in parallel. The main thread then assures the completeness known from breadth-first search whereas all other threads do a fast parallel depth-first search. As the structure of the search tree is shared between both evaluation strategies, the breadth-first search then partly encounters subtrees, which are already evaluated to weak head normal form.

A first approach might be implementing this new strategy *bfsParallel*3 in the following way.

```
bfsParallel3 t = splitAll1 t `par` bfsSearch t
```

This would not evaluate the list completely, as *rpar* only does an evaluation to weak head normal form. We need to evaluate the whole list skeleton, but we do not care for the elements of the list. We write a function *evalList*, which evaluates the list skeleton, but ignores all elements and in the end always returns ().

```
evalList [] = ()
evalList (_ : xs) = evalList xs
```

Since *evalList* completely ignores the elements, it does not keep the list in memory. The parts of the search tree needed for the traversal will be kept in memory by the breadth-first search.

Applying *evalList* to the result of *splitAll*1 gives the wanted result.

```
bfsParallel3 :: SearchTree a → [a]
bfsParallel3 t = evalList (splitAll1 t) `par` bfsSearch t
```

Implementing this function in the *Eval* monad would be slightly more difficult. A similar implementation might look as follows:

```
bfsParallel3 t = runEval $ do
  _ ← rpar (evalList (splitAll1 t))
  return (bfsSearch t)
```

However, this is not equivalent to the definition above. The result of *rpar* is ignored and thus the created spark may be deleted with its evaluation being aborted. Therefore, the result of the list needs to be kept in memory until the search is completed or itself marked as dead. One way to achieve this is to change the result of *evalList* to a list and append the result of the spark to the end of the list.

```
evalList [ ] = [ ]
evalList (_ : xs) = evalList xs

bfsParallel3 t = runEval $ do
  d ← rpar (evalList (splitAll1 t))
  return (bfsSearch t ++ d)
```

The evaluation of the spark may not be complete at the end of the evaluation of *bfsSearch t*, for instance when the spark is not taken from the spark pool. Then the evaluation of the spark will still be forced at its result is needed for the evaluation of the result list. The implementation above does not have this problem.

The main difference between *bfsParallel2* and *bfsParallel3* is that *bfsParallel2* creates a spark for each subtree and therefore each spark only evaluates the top level of each choice. In *bfsParallel3* the sparks only create a spark for the right subtree and continue evaluating the left branch down to its leaf.

4.4. Bag of Tasks

GHC's Haskell runtime provides support for deterministic parallelism (section 2.2.1), which we have used successfully to speed up the search (section 4.3). However, the deterministic strategies have certain limitations when we have no need for all values in the search tree. Consider the following, taken from the implementation of the strategy *splitAll1*.

```
splitAll1 :: SearchTree a → [a]
splitAll1 None     = [ ]
splitAll1 (One x) = [x]
splitAll1 (Choice l r) = runEval $ do
  rs ← rpar (splitAll1 r)
  ls ← rseq (splitAll1 l)
  return (ls ++ rs)

someValue :: SearchTree a → a
someValue = head ∘ splitAll1
```

While the third rule of *splitAll*1 allows the parallel evaluation of the right subtree, it always waits for the left subtree to provide a value first. On one hand this ensures that the order of the elements of the resulting list is identical to the order of the elements returned by the depth-first search; the value returned by *someValue* is always the same for a given search tree. On the other hand we possibly have to wait for a long time on the evaluation of *someValue*, even if there are values that already have been evaluated in parallel. Often, non-deterministic problems do not depend on the order of results, so one would prefer a strategy that returns the values of an expression as quickly as possible, but in an arbitrary order.

After describing the general idea (section 4.4.1), we can define multiple search strategies similar to the sequential strategies from section 4.2. We define strategies analogue to the depth-first search (section 4.4.2) and the breadth-first search (section 4.4.3) next and present the implementation of the bag of tasks library in section 5.2.

4.4.1. General idea

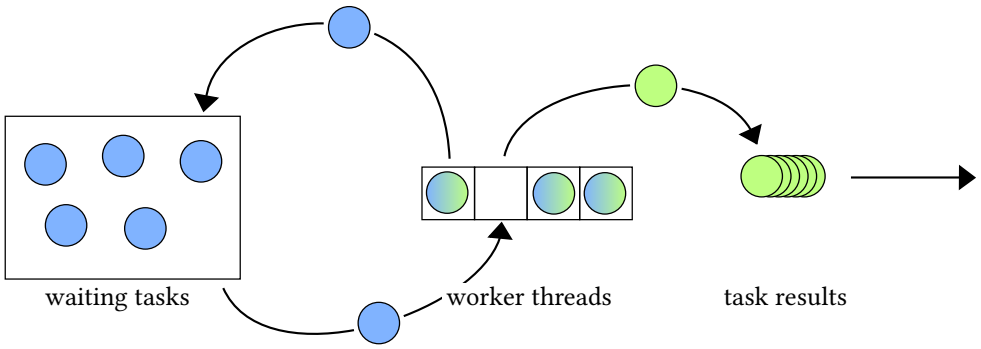


Figure 4.12.: A bag of tasks with waiting tasks (blue disks), task results (green disks) and worker threads (squares).

The solution for this problem is a *bag of tasks* approach. In this approach, there is a buffer of tasks and a "gang" of *worker threads* working on these tasks. Each worker takes a task out of the buffer and processes it. During the processing, the worker adds additional tasks into the bag and optionally writes back a *task result* at the end. After processing the task, the worker takes the next task from the buffer of *waiting tasks*. When no values are required anymore, it needs to be possible to stop the bag: the threads are terminated and the remaining tasks can be thrown away.

A *high level* interface for the bag of tasks provides a function to start a task with a given list of initial tasks and an action to process the results sequentially:

$$runTaskBag :: [Task\ a\ (Maybe\ a)] \longrightarrow IO\ [a]$$

runTaskBag initialises a new bag of tasks and starts a gang of worker threads as described above. The worker threads immediately start to process the given tasks. *runTaskBag* returns a lazy list of all thread results. As the list has an arbitrary order, it is returned in the *IO* monad. The number of worker threads started is equal to the number of system threads (see section 2.2.3) of the Haskell runtime system.

Task instanciates both *Monad* and *MonadIO* and therefore has the function *liftIO* to perform arbitrary *IO* actions in the *Task*. Additionally there is a function *addTask* to add new tasks to the bag.

$$addTask :: Task\ r\ (Maybe\ r) \longrightarrow Task\ r\ ()$$

The action provided as the first argument of *runTaskBag* may or may not return a value. If it returns a value, this value is written back as a result.

The bag and its threads will be terminated when the list of values becomes garbage collected. However, terminating the running threads may be impossible in rare cases, see section 5.1.1.

Later we will see that it is profitable to allow different kinds of task buffers. Sometimes we would like to have the first item added to the buffer to be processed first (*first in, first out*) and sometimes the need the last item added to the buffer to be processed first (*last in, first out*). Therefore, we can choose the buffer type in the initialisation of the task bag with a value of type *BufferType*.

$$\textbf{data}\ BufferType = Queue \mid Stack$$

The value *Queue* selects tasks buffers with *first in, first out* order whereas the value *Stack* selects a *last in, first out* order.

This type becomes the additional argument type of *runTaskBag*.

$$runTaskBag :: BufferType \longrightarrow [\,Task\ a\ (Maybe\ a)\,] \longrightarrow IO\ [\,a\,]$$

4.4.2. Depth-First Search

```
dfsTask :: SearchTree a → Task a (Maybe a)
dfsTask None     = return Nothing
dfsTask (One v)  = return (Just v)
dfsTask (Choice l r) = do
  addTask (dfsTask r)
  dfsTask  l
```

Figure 4.13.: The depth-first task.

Using a bag of tasks implementation as presented in section 4.4.1, we can now define strategies similar to those defined in section 4.3. The strategy *splitAll*1 resembles the sequential depth-first search and adds a new spark for the right subtree at each choice constructor while

evaluating the left subtree. Analogue to *splitAll*1 we define a task *dfsTask* with the same behaviour in figure 4.13. If the task is applied to a *None* it does not yield a result; if there is a *One* constructor in the search tree, we found a result and return this result; at a *Choice* constructor, we add a new task for the right subtree, which may be taken by another worker thread, and continue to evaluate the left subtree immediately.

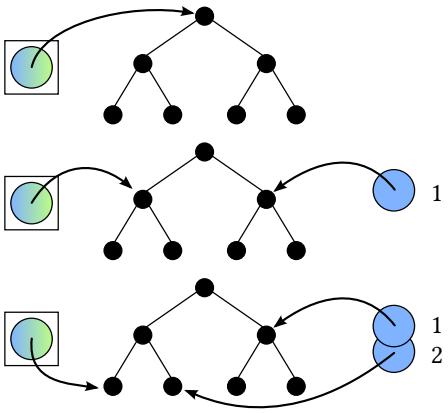


Figure 4.14.: A *dfsTask* searching values in a search tree with one worker thread.

Figure 4.14 visualises the processing of a search tree with the *dfsTask*. Whether or not the search resembles a real depth-first search, depends on the kind of the task buffer. Using a stack as task buffer leads to a normal depth-first search; thus, we call this strategy *dfsBag*.

```
dfsBag :: SearchTree r → IO [r]
dfsBag tree = runTaskBag Stack [dfsTask tree]
```

When we take a queue for the task buffer, we get a search strategy which follows each branch to its leftmost leaf first but then continues with the topmost right subtree. We call this strategy *fdfsBag*.

```
fdfsBag tree = runTaskBag Queue [fdfsTask tree]
```

Reducing the Number of Tasks

As the custom-built bag of tasks framework will have a more significant overhead in comparison to the sparks, which are built-in in GHC's runtime system, it will be even more important to minimise the number of tasks. For depth-first search, we use the same techniques as above in section 4.3.3. Therefore, we get a strategy called *dfsBagLimit* limiting the depth to which the evaluation of the search tree is parallelised, a strategy called *dfsBagAlternating* splitting the evaluation at every n-th choice constructor, and the strategies *dfsBagRight* and *dfsBagLeft* evaluating only the right/left subtrees in parallel.

All these strategies require to return more than one value per task. With the current bag of tasks, this is not possible, as tasks are expected to have the type *Task a* (*Maybe a*). In order to get tasks with multiple results, it is possible to alter this to tasks having the following type:

```
Task a [a]
```

However, this requires to collect all the results and return it at the end in one list. Thus, results are not returned immediately, which delays further processing. Another approach is to provide a function *writeResult* to return results in the middle of a running task.

```
writeResult :: r → Task r ()
```

To illustrate the use of *writeResult*, we consider the implementation of the task for *dfsBagLimit*.

```
dfsLimitTask :: Int → SearchTree a → TaskIO a (Maybe a)
dfsLimitTask n None     = return Nothing
dfsLimitTask n (One x) = return (Just x)
dfsLimitTask 0 t@(Choice _ _) = do
  mapM_ writeResult (dfsSearch t)
  return Nothing
dfsLimitTask n (Choice l r) = do
  addTaskIO (dfsLimitTask (n - 1) r)
  dfsLimitTask (n - 1) l
```

Similar to *splitLimitDepth*, we need an additional integer argument for the *split depth*. Once the split depth is zero, we evaluate the search tree sequentially with *dfsSearch* and write back the results lazily.

The implementations of *dfsAlternatingTask*, *dfsRightTask*, and *dfsLeftTask* are similar and therefore omitted here.

4.4.3. Breadth-First Search

The general idea of breadth-first search is to evaluate the search tree level by level. In contrast to the depth-first search tasks, the tasks for breadth-first search may only evaluate subtrees to weak head normal form. After evaluating one subtree, the computation has to continue with the next subtree of the same level until the complete level has been evaluated. To implement this behaviour, we can use tasks which only do the evaluation of one subtree to weak head normal form and then add the possible children as new tasks. This task is implemented in figure 4.15.

Figure 4.16 shows the traversal of a search tree with the *bfsTask*. The single worker is shown as a box at the left of the search tree whereas the task buffer is shown at the right of the tree. The numbers next to the task buffer indicates the order in which the tasks have been added to the buffer. To obtain a breadth-first search, it is necessary to take a queue for the task buffer. Having a stack buffer instead would make a significant difference after the fourth step. The worker would then get task 3, which would also result in a depth-first search strategy. As this

```
bfsTask :: SearchTree a → Task a (Maybe a)
bfsTask None        = return Nothing
bfsTask (One v)     = return (Just v)
bfsTask (Choice l r) = do
  addTask (bfsTask r)
  addTask (bfsTask l)
```

Figure 4.15.: The breadth-first task.

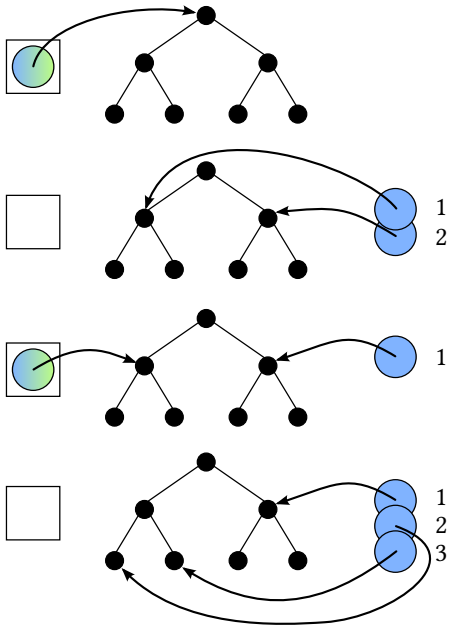


Figure 4.16.: A *bfsTask* searching values in a search tree with one worker thread.

would generate significantly more accesses to the thread buffer compared to the depth-first strategy above, we did not implement this depth-first search.

4.5. Fair Search

Most parallel search strategies use parallelism to reduce the time needed for the evaluation of an expression. In contrast to these, the *fair search* only focuses on being complete which often even leads to a drawback in terms of evaluation time. Total completeness cannot be achieved with sequential search strategies like breadth-first search. A search tree of a non-deterministic computation can contain an infinite deterministic computation, for instance a loop, in each of its branches (see section 3.2). This leads to the need of evaluating all branches simultaneously.

All fair search strategies presented in this section are implemented by starting independent computation threads, similar to the concurrent implementation of Curry in Java presented 1999 by Hanus and Sadre [9]. *Preemptive multitasking* interrupts all these threads regularly and selects another thread to be run next. It thereby ensures that each thread runs at some point, and furthermore that all values in the search tree are computed. These strategies are called fair because they do not favour one of two subtrees. The presented strategies differ in the way the computation threads communicate with each other.

4.5.1. Primitive Fair Search

The first, primitive implementation of a fair search strategy, *fairSearch*1, evaluates the complete search tree in parallel using threads as shown in figure 4.17. The evaluation splits into two threads each time a *choice* occurs in the search tree. The right subtree is then evaluated in the new thread whereas the old thread continues to evaluate the left subtree. If no value is found in the search tree, the calculation terminates with no result; any results found are written into a *Chan* (section 2.2.2).

```
searchThread :: MVar ExecutionState → ResultChan → SearchTree a → IO ()
searchThread threadVar resultChan tree =
  case tree of
    Choice l r → do
      startSearchThread threadVar resultChan r
      searchThread      threadVar resultChan l
    None       → return ()
    One x      → writeChan resultChan (Value x)
```

Figure 4.17.: Haskell code of the function *searchThread*.

As the *fair search* is especially useful when applied to search trees with infinite deterministic computations, where no search can find all possible values, the user would demand only a

certain number of results and the calculation has to be stopped afterwards. To stop all running threads when no value is demanded anymore, we need to keep a list of threads working on the current search tree. This is stored in the *MVar threadVar*. The threads stored in this *MVar* may be stopped later when no other value will be demanded. To prevent that a thread is started during the killing of the already existing threads, we have to synchronise the killing with the starting of the threads. Then, this new thread may not be registered in the *threadVar* yet. To preclude this, the *threadVar* contains an additional flag to signal that the evaluation has been aborted already. Therefore the type of the *threadVar* is *ExecutionState*.

```
data ExecutionState = Stopped
                    | Executing [ThreadId]
```

If the value stored in the *threadVar* is *Stopped*, no new threads are allowed to be started. Being in the state *Executing* means that the computation is still ongoing. Its argument is the list of threads evaluating the tree. Being in the state *Executing* but having no threads left means, that all values of the tree have been calculated.

The function *startSearchThread* starts a new thread as shown in figure 4.18. The function has to make sure that it is allowed to start a new thread before doing so. After starting the thread, it has to be added to the list of new threads in the *threadVar*. Removing the thread from this list has to be done after reading the possible result from this thread, because the list of threads is used to determine if we can expect further results from the evaluation. This is why it is not possible to do this in the thread itself. The *ThreadStopped* is sent at the end of the execution of the new thread through the result channel. Removing the threads from the list is done from another thread, after the message *ThreadStopped* is read from the result channel.

```
startSearchThread :: MVar ExecutionState
                  → Chan (ThreadResult a)
                  → SearchTree a
                  → IO ()
startSearchThread threadVar chan tree = do
  executeState ← takeMVar threadVar
  case executeState of
    Stopped       → do
      putMVar threadVar executeState
    Executing tids → do
      newTid ← forkIO $ do
        tid ← myThreadId
        searchThread threadVar chan tree
        writeChan chan (ThreadStopped tid)
      putMVar threadVar (Executing (newTid : tids))
```

Figure 4.18.: Haskell code of the function *startThread*.

The code performed in the main search thread can be seen in figure 4.19. The search thread beginning at the root of the search tree is started by the main thread after initializing the

threadVar and the result channel *resultChan*. This root thread computes all values in the search tree and forks new threads where necessary. The results written into the result channel are read by the function *handleResults*. The execution of *handleResults* is deferred lazily using *unsafeInterleaveIO*:

```
unsafeInterleaveIO :: IO a → IO a
```

The *IO* action given as the argument of *unsafeInterleaveIO* is executed once the value type *a* is demanded. Deferring the execution of *handleResults* makes it possible to return from *fairSearch*1 immediately even if no values have been computed yet.

```
fairSearch1 :: SearchTree a →
fairSearch1 tree = do
  threadVar  ← newMVar (Executing i [ ])
  resultChan ← newChan
  startSearchThread threadVar resultChan tree
  unsafeInterleaveIO (handleResults threadVar resultChan)
```

Figure 4.19.: Haskell code of the main function of the fair search.

The function to read the results from the channel can be seen in figure 4.20. It immediately tries to read a value from the channel with *readChan*, which blocks until there are any values in the channel. Once a value is available, the execution continues and checks whether the read result is a *ThreadStopped* notification or a value. A value is returned as the head of the value list after deferring the execution of *handleResults*. If the *ThreadStopped* notification is read, the given thread identifier has to be removed from the list of threads with *modifyMVar*.

```
modifyMVar :: MVar a → (a → IO (a, b)) → IO b
```

The function *removeThread* only transforms one *ExecutionState* into another one that does not contain the given *ThreadId* anymore. *modifyMVar* takes an *IO* action with a pair of results. One of these is written back into the mutable variable and the other value is returned by *modifyMVar* itself for further inspection. We need the whole *ExecutionState* and thus return this two times.

At this point, we need to know if there still are threads left to compute further values. If there are further threads and the *ExecutionState* is *Executing*, we have to continue reading values from the result channel. If there are no other threads left or the value is *Stopped*, no value can be computed anymore and we reached the end of the list.

To make the definition of the fair search strategy complete, we need a function to stop the evaluation. This function is called *killThreads* and is called with the technique explained in section 5.1. The implementation of *killThreads* can be seen in figure 4.21. *killThreads* first takes the *threadVar* so that other threads will not be started in the meantime. If the state is *Stopped*, there is nothing left to do. If it is still in the state *Executing*, there are a number of threads that have to be killed. This is being done in separate threads as *killThread* blocks until it was possible to terminate the thread. After stopping the threads, the *threadVar* is set to

Stopped. Therefore, still running threads are not able to start new threads as this requires the state *Executing*.

Because of the shared state in the variable *threadVar*, which has to be changed at each starting or stopping of a thread, a lot of synchronisation happens between the threads. The following sections describe slightly modified search strategies trying to reduce the synchronisation points.

4.5.2. Fair Search with Chained Threads

The idea to reduce the synchronisation between the threads is not to have a central synchronisation variable which all threads synchronise on. Instead, each thread only communicates with the thread that started it and the threads having been started by this thread. Figure 4.22 shows the communication during the evaluation of a sample tree. The arrows pointing at the beginning of each threads lifetime indicate the forking of a new thread. All other arrows are asynchronous messages.

In contrast to the fair search strategy above, a thread which discovers a *Choice* constructor starts two child threads to evaluate one of the subtrees each. Then, the parent thread starts to listen on a channel to receive the messages of its children. There are two important messages being sent from the child threads to its parents, *Value x* and *ThreadStopped*. *Value x* is sent once a thread discovered a *One* constructor, whereas *ThreadStopped* is sent at the end of its execution. When a thread receives a *Value* message, it propagates this value to its own parent. The life of a thread ends once a thread finds a *None* leaf in the tree or once it received a *ThreadStopped* message from all of its children.

This concept still misses the possibility of stopping the threads when no further values will be demanded. As there is no central point at which all threads are registered, this can not be done at once by the main thread; instead, it has to be done via the tree structure of the threads. At the time of aborting a computation, some threads are waiting on a message coming through its channel while others are evaluating parts of the search tree. The message to stop the computation may be sent to the waiting threads via their message channels. The threads evaluating the search tree do this evaluation in pure code and therefore cannot be reached using a channel. A way to reach these threads in pure code is sending an exception, similar to the *ThreadKilled* exception sent by *killThread*.

For the sake of a simple design, we choose to send an exception to all threads. This search strategy is called *fairSearch*2 as it is a variant of the *fairSearch*1 strategy above. Here, we are not interested in terminating the threads with a *ThreadKilled* exception; threads, which already started child threads, have to forward this exception to their children to kill the whole search process. This is why we define the new exception *Stop*.

```
data Stop = Stop
  deriving (Typeable, Show)
instance Exception Stop
```

```
handleResults :: MVar ExecutionState → Chan (ThreadResult a) → IO [a]
handleResults threadVar resultChan = do
  result ← readChan resultChan
  case result of
    Value a → do
      as ← unsafeInterleaveIO (handleResults dummyRef threadVar resultChan)
      return (a : as)
    ThreadStopped tid → do
      state ← modifyMVar threadVar (return ∘ (λa → (a, a)) ∘ (removeThread tid))
      case state of
        Executing (_ : _) →
          handleResults dummyRef threadVar resultChan
        _ →
          return [ ]
removeThread :: ThreadId → ExecutionState → ExecutionState
removeThread tid Stopped          = Stopped
removeThread tid (Executing tids) =
    Executing (delete tid tids)
```

Figure 4.20.: Reading results from the value channel.

```
killThreads :: MVar ExecutionState → IO ()
killThreads threadVar = do
  executeState ← takeMVar threadVar
  case executeState of
    Stopped →
      return ()
    Executing tids →
      mapM_ (forkIO ∘ killThread) tids
  putMVar threadVar Stopped
```

Figure 4.21.: Code to stop the search threads.

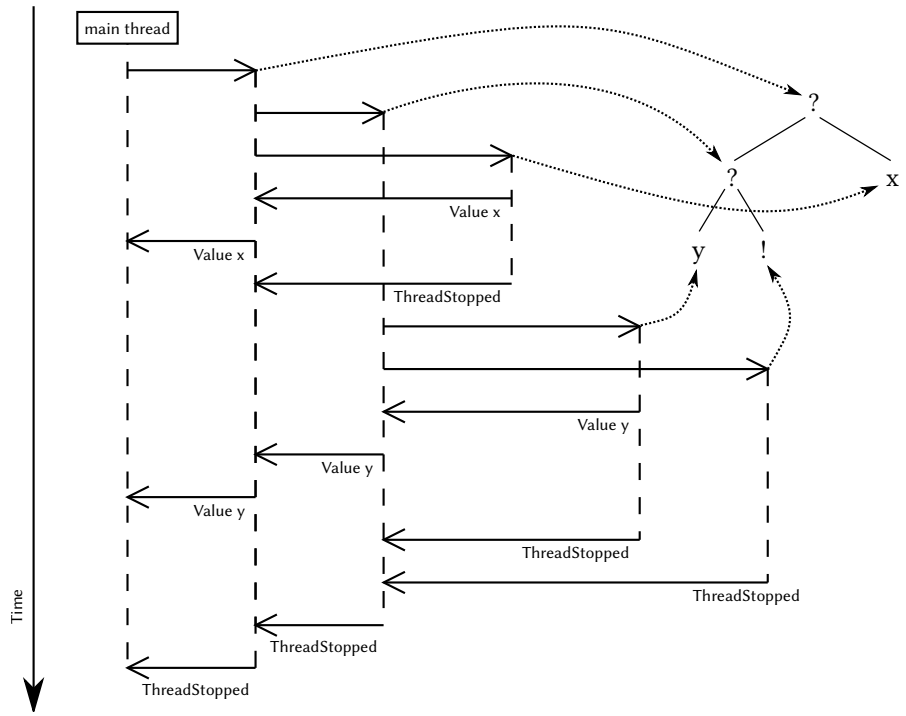


Figure 4.22.: Chained communication between search threads.

Furthermore, we have to make sure that this exception is not being received at certain points, like when starting new children. To avoid this, it is possible to mask all exceptions in a part of the evaluation.

$$uninterruptibleMask\_ :: IO\ a \longrightarrow IO\ a$$

At certain points, we can safely unmask the exception.

$$forkIOWithUnmask :: ((forall\ a \circ IO\ a \longrightarrow IO\ a) \longrightarrow IO\ ()) \longrightarrow IO\ ThreadId$$

The function *forkIOWithUnmask* starts a new thread executing the given *IO* action, but in contrast to *forkIO* it provides an unmasking function to this *IO* action. This unmasking function is used during the evaluation of the search tree and while waiting on possible values from the channel.

The evaluation of the search tree happens in *searchThread*, which has the unmasking function as its first parameter *unmask*. The code of *searchThread* can be seen in figure 4.23. To catch the exception *Stop*, thrown during the evaluation of pure code, we have to demand the evaluation of the search tree to weak head normal form explicitly with the function *evaluate*.

$$evaluate :: a \longrightarrow IO\ a$$

The exception can then be caught with *catch* which takes the *IO* action to catch the exception in and the exception handler.

$$catch :: Exception\ e \Rightarrow IO\ a \longrightarrow (e \longrightarrow IO\ a) \longrightarrow IO\ a$$

When the exception has been catched during this evaluation, we want to stop the computation in this thread, which is similar to what we do in the case of a failure in the search tree.

Having evaluated a choice constructor, the search thread creates the new message channel and starts its children with *forkIO*. The identifiers of these threads have to be saved, since this thread terminates once all of these children sent their *ThreadStopped* message through the newly created channel.

Listening to children messages happens in the function *listenChan*, which can be seen in figure 4.24. *listenChan* has to perform three tasks: propagating result values back to its parent thread, forwarding the *Stop* exception to abort the computation, and waiting for all children having stopped their execution. Catching the *Stop* exception is only done while reading values from the channel. The result from this reading process is then either *Nothing*, in the case of an exception, or *Just s* with *s* being the message read from the channel. Forwarding the *Stop* exception to the child threads is done by the exception handler itself.

Reading from the parent channel for the read node is all the main thread has to do to collect all values; all values are forwarded to this in the end. At some point it receives a *ThreadStopped* message. Then, all values have been read from the channel, because *Value* messages are always sent before *ThreadStopped* and the threads forward these messages in the same order as they have been sent. Now, killing all threads is easy: it just sends the *Stop* signal to the root thread.

```
searchThread :: (forall a ∘ IO a → IO a)
             → Chan (ThreadResult b)
             → SearchTree b
             → IO ()
searchThread unmask parent tree = do
  etree ← unmask (evaluate tree) `catch` (λStop → return None)
  tid   ← myThreadId
  case etree of
    None       →
      writeChan parent (ThreadStopped tid)
    One v      → do
      writeChan parent (Value v)
      writeChan parent (ThreadStopped tid)
    Choice l r → do
      chan ← newChan
      t1   ← forkIO (searchThread unmask chan l)
      t2   ← forkIO (searchThread unmask chan r)
      listenChan unmask [t1, t2] chan parent
```

Figure 4.23.: The main search thread function of *fairSearch*2.

```
listenChan :: (forall a ∘ IO a → IO a)
           → [ThreadId]
           → Chan (ThreadResult b)
           → Chan (ThreadResult b)
           → IO ()
listenChan _          [] _      parent = do
  tid ← myThreadId
  writeChan parent (ThreadStopped tid)
listenChan unmask ts chan parent = do
  ans ← unmask (liftM Just (readChan chan)) `catch` (λStop → do
    mapM_ (λt → throwTo t Stop) ts
    return Nothing)
  case ans of
    Just (ThreadStopped t) → listenChan unmask (delete t ts) chan parent
    Just value → do
      writeChan parent value
      listenChan unmask ts chan parent
    Nothing    → return ()
```

Figure 4.24.: The code to listen for messages from the children of the search thread.

killThreads root = forkIO (*throwTo root Stop*)

On one hand, using this technique, it is not necessary to have a globally readable state. On the other hand, we need to create more threads and also more channels for the communication between the threads. Values have to be sent through a possibly long chain of threads until they are read from the main thread. However, the idea of using exceptions for the communication between threads will be picked up again in the next section.

4.5.3. Using Exceptions for Communication between Search Threads

In the fair search strategy *fairSearch*2 presented above, large parts of time and memory are spent on forwarding the result values. This is why the third fair search approach uses a mixture of the strategies presented above. Values are written into a common channel as in *fairSearch*1, but the child threads are managed by the threads that started them like in *fairSearch*2. Rather than letting certain threads wait on the results or the stopping of others like *fairSearch*2, the threads continue evaluating the left subtree after finding a choice constructor; only the right subtree is evaluated by a newly started child thread. Figure 4.25 shows the communication between the threads during the evaluation of the same search tree that has been used for *fairSearch*2 in figure 4.22. In addition to the saved overhead of forwarding the messages, we also do not have to create half of the threads.

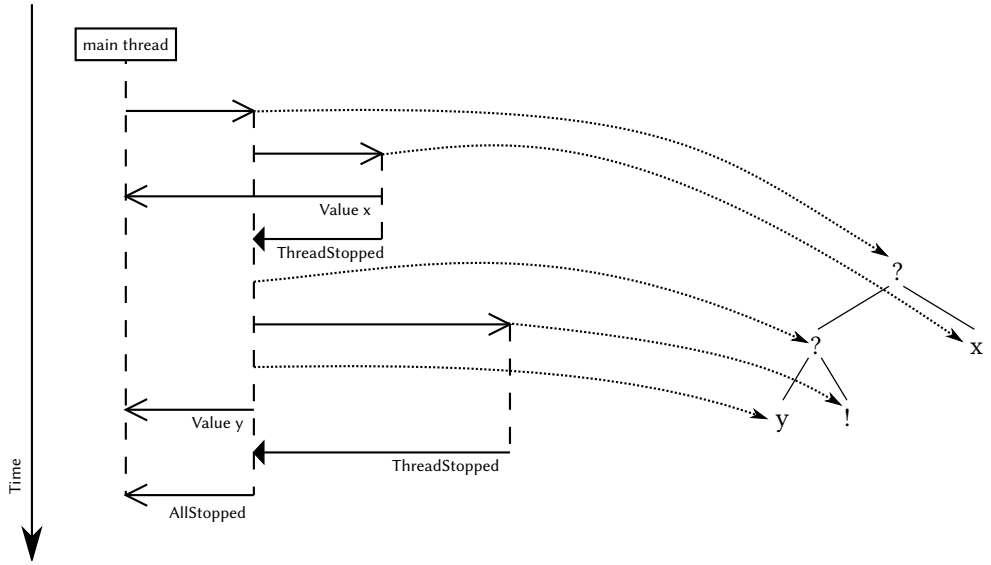


Figure 4.25.: Chained communication between search threads.

In contrast to *fairSearch*2, the number of created search threads per thread is not limited by a fixed number: it depends on the height of the search tree. As a result we will have

to wait on a significantly larger number of child threads. The main problem at managing the child threads is that the threads can not wait on a channel to receive *ThreadStopped* messages. This is because they have to continue evaluating their own part of the search tree. Also looking for values in such a channel is unfeasible, because it would require a function *isEmptyChan*.

$$isEmptyChan :: Chan\ a \longrightarrow IO\ Bool$$

This function does exist, but it does not work correctly and therefore is deprecated [28]. The alternative to receiving the *ThreadStopped* messages directly or after a short interval is reading these messages once the thread would stop itself.

To save memory, we decide against this in favour of communication using exceptions. Exceptions can be received directly, even while executing pure code. Furthermore, it is possible to catch any exception that has been thrown up to a certain point of time while continuing normally when no exception has been thrown. In figure 4.25 arrows with solid arrow heads represent communications using exceptions whereas communication through a channel is represented by arrows with open arrow heads.

When an exception is caught during the evaluation of pure code, the results seem to be lost as the value of the exception handler is returned instead. However, in Haskell intermediate results of pure evaluations are often kept in memory as long as a reference to the expression exists. Thus, the time lost for thereby aborted computations is negligible.

Apart from the usage of exception to communicate between the search threads, figure 4.25 shows that there is another significant difference between this strategy and the strategies presented before: the first search thread, starting at the root of the search tree, has to write *AllStopped* into the result channel while all other threads throw the exception *ThreadStopped* to their parent at the end of their execution. Nevertheless, because the rest of the behaviour of the threads is identical, the code for *searchThread* in figure 4.26 is used for all search threads. In comparison to the other *searchThread* implementations, it has an additional parameter containing the action that has to be performed at the end of its lifetime.

The time frames in which messages, here exceptions, are received are identical to these of *fairSearch*2, but here it is always possible to receive two types of exceptions: *Stop* and *ThreadsStopped*. The function *listenExceptions* has the same job as *listenChan* in the implementation of *fairSearch*2. It waits for both *Stop* and *ThreadStopped* messages. In comparison to *listenChan*, there is no channel to block on. This makes it necessary to block artificially on a newly generated *MVar*.

The implementation of *fairSearch*3 combines the strengths of both search strategies *fairSearch*1 and *fairSearch*3. It needs the minimal number of threads and writes back values directly, like *fairSearch*1, but does not have the synchronisation overhead required to manage the threads centrally.

```
data EvalResult a = Finished (SearchTree a)
                  | ReceivedStop
                  | ReceivedStoppedSearch ThreadId
searchThread :: (forall a . IO a -> IO a)
             -> IO ()
             -> Chan (ThreadResult b)
             -> [ThreadId]
             -> SearchTree b
             -> IO ()
searchThread unmask end chan threads tree = do
  r <- unmask (liftM Finished (evaluate tree))
    `catch` (\Stop                  -> return ReceivedStop)
    `catch` (\(ThreadStopped tid) -> return (ReceivedStoppedSearch tid))
  case r of
    ReceivedStop                ->
      stopChildren end threads
    ReceivedStoppedSearch tid ->
      searchThread unmask end chan (tid `delete` threads) tree
    Finished t                  ->
      case t of
        None  -> listenExceptions unmask end threads
        One v -> do
          writeChan chan (Value v)
          listenExceptions unmask end threads
        Choice l r -> do
          tid   <- myThreadId
          child <- forkIO (searchThread unmask (notifyStopped tid) chan [] r)
          searchThread unmask end chan (child : threads) l

notifyStopped tid =
  myThreadId >>= (throwTo tid) . ThreadStopped

stopChildren end threads = mapM_ (\t -> throwTo t Stop) threads >> end

listenExceptions :: (forall a . IO a -> IO a) -> IO () -> [ThreadId] -> IO ()
listenExceptions _      end []      = end
listenExceptions unmask end threads = do
  m <- newEmptyMVar
  unmask (void $ takeMVar m)
    `catch` (\Stop                  ->
      stopChildren end threads)
    `catch` (\(StoppedSearch tid) ->
      listenExceptions unmask end (tid `delete` threads))
```

Figure 4.26.: The search thread function of *fairSearch*3.

5. Technical Details

This chapter is about patterns and technologies that are not directly related to Curry and parallel search strategies. Thus, they had to be discovered to implement various strategies. Techniques to stop parallel evaluation (section 5.1) are used in the implementation of the *bag of tasks* (section 5.2) and the *fair search strategies* (section 4.5).

5.1. Stopping Parallel Evaluation

To evaluate an expression in parallel to the main execution, a simple approach is to fork another thread and evaluate the operation in the new thread. When the forked operation has been evaluated to weak head normal form, it may be put into an *MVar*. When the main thread requires the value of the expression, it *forces* its evaluation. This *MVar* will be read and therefore the thread blocks until the evaluation is complete. We call this *MVar* a *future* because it represents a value which will be available in the future. Figure 5.1 shows a function to start the evaluation of an expression in parallel. However, at the time of calling *parallel*, it

```
type Future a = MVar a

newFuture :: a → IO (Future a)
newFuture e = do
  m ← newEmptyMVar
  _ ← forkIO (e `seq` putMVar m e)
  return m

force :: Future a → IO a
force f =
  readMVar f
```

Figure 5.1.: A primitive implementation of a *future*.

may be unknown if the value of ε will be required later, but we still want to start its evaluation early to reduce the time of waiting. Using the approach presented above has an important shortcoming in this use case: the forked thread would continue evaluating the expression even if we know that its value will not be needed anymore.

5.1.1. Stopping Threads Explicitly

A solution for this problem would be to stop the thread explicitly. In Haskell, stopping a thread can be done using the function *killThread*.

```
killThread :: ThreadId → IO ()
```

This function raises the *asynchronous exception ThreadKilled* to the thread specified by the given *ThreadId*. The *ThreadId* is returned by the function forkIO, so we can add it to the type *Future* easily.

```
type Future a = (MVar a, ThreadId)
newFuture :: NFData a ⇒ a → IO (Future a)
newFuture e = do
  m ← newEmptyMVar
  tid ← forkIO (e `seq` putMVar m e)
  return (m, tid)
stopFuture :: Future a → IO ()
stopFuture (_, tid) = killThread tid
```

Figure 5.2.: Stopping the evaluation explicitly with *killThread*. The function *force* is omitted, because it can be adapted trivially.

Figure 5.2 shows how to stop the evaluation of the future explicitly with the function *stopFuture*. The implementation of the stoppable future is easy to read and seems to be correct. Below, we explain some tests of our implementation. The following function defines a list of all prime numbers by filtering all multiples of found primes from the list of all greater natural numbers.

```
primes = primes' 2
  where
    primes' n = n : filter (λm → m `mod` n ≠ 0) (primes' (n + 1))
```

As there is an infinite number of primes, the list *primes* is also infinite. Thus, the evaluation of the expression *last primes* does not terminate, but this does not keep us from starting a future to evaluate it to weak head normal form. Unfortunately, this evaluation will never stop, so we call *stopFuture* on the future which stops the evaluation *immediately*.

Now consider the function *ones*, which defines an infinite list of integers as well: the list containing an infinite number of the value 1.

```
ones = 1 : ones
```

Again, the evaluation of the expression *last ones* does not terminate and so does the evaluation of a corresponding future. Though this time a call of *stopFuture* does not stop the future and the call does not terminate itself.

In today's GHC runtime system (version 7.8.1), certain evaluations cannot be stopped. To stop the thread our implementation uses *killThread* which is itself using *throwTo* to throw an asynchronous exception to the specified thread. In GHC, asynchronous exceptions can only be received at *safe points* [26]. A safe point is wherever memory allocation occurs [23]. In general, memory allocation is rather frequent and normal code without memory allocation is very rare, yet the above example shows that such code exists. We depend on the informed programmer here to modify her programs with the objective of introducing memory allocation.

Code which does not do any memory allocation can even lead to further problems. For instance, preemptive scheduling is only run at safe points, as well. Running the future evaluating the expression *last ones* in a Haskell runtime with only one system thread would cause the future thread to claim this system thread forever. Therefore the main program will never run again and the call to *killThread* will never be reached.

Another in our case unwanted behaviour may be that *killThread* and thus *stopFuture* blocks until the signal is received. This can be fixed easily by forking into another thread to call *killThread*.

```
stopFuture :: Future a → IO ()
stopFuture (_, tid) = forkIO (killThread tid)
```

5.1.2. Using Finalisers on Weak Pointers

Though the explicit stopping of the thread is a solution to the problem of wasted resources, it reveals a lot of the implementation of the *Future*[1] and it requires a deep knowledge of the surrounding program from the programmer.

Thus, we take another approach into regard. Our implementation uses Haskell's *weak pointers*, which were presented by Jones, Marlow, and Elliott [15]. You can find a more complete explanation of Haskell's weak pointers in section 2.2. Haskell's weak pointers have the following interface:

```
data Weak v
mkWeak     :: k → v → Maybe (IO ()) → IO (Weak v)
deRefWeak  :: Weak v → IO (Maybe v)
```

The function *mkWeak* creates a weak reference from the *value* given as the second argument. The first argument is the *key* to the weak reference. As long as the key is *reachable*, calling *deRefWeak* on the weak reference will return the value. When the key becomes unreachable the weak reference may be finalised. Finalisation means that deRefWeak is made to return *Nothing* and the operation given by the third argument of *mkWeak*, called the finaliser, will be run (see [18]).

[1] For instance that it is implemented using a thread that has to be stopped.

In our primitive future implementation of figure 5.1 we may need the result of the evaluation as long as we keep the future itself, which is an *MVar*, reachable, so we may use this as our key. To make sure the *MVar* becomes unreachable when it is unreachable from the main thread it has to be the only thread having a direct reference to it. Thus, it has to be unreachable from all other threads including the thread evaluating the expression. To be still able to write the result back into the *MVar*, we can use a weak reference to it from the forked thread. The finaliser would then have to kill the thread.

```
type Future a = MVar a

newFuture :: NFData a => a -> IO (Future a)
newFuture e = do
  m <-     newEmptyMVar    - MVar to hold the resulting value
  t <-     newEmptyMVar    - MVar to hold the ThreadId
  w <-     mkWeakMVar m $ do
    child <- readMVar t
    void (forkIO (killThread child))
  child <- forkIO $ do
    n <- e `seq` deRefWeak w
    case n of
      Nothing -> return ()
      Just k  -> putMVar k e
  putMVar t child
  return m
```

Figure 5.3.: Using a weak reference to an *MVar* to stop a thread.

The implementation in figure 5.3 shows two characteristics. We need another *MVar t* to store the thread id, because this is not available at the time the weak reference is created and we need the weak reference in the thread.
Additionally we do not use *mkWeak* directly but the function *mkWeakMVar*.

```
mkWeak     :: k -> v -> Maybe (IO ()) -> IO (Weak v)
mkWeakMVar :: MVar a -> IO () -> IO (Weak (MVar a))
```

mkWeakMVar is a bit different from *mkWeak* as the given *MVar* is, in a certain way, both the value and the key of the weak reference. This function exists because *mkWeak* does not work as expected for certain built in types like *MVar*, as explained in the GHC ticket [19]. Using *mkWeak* on a *MVar* would attach the weak reference to the box *MVar* and not the primitive type *MVar#* underneath. The optimisation may eliminate this box and therefore the key is unreachable. *mkWeakMVar* was introduced in reaction to this ticket and fixes the problem by attaching the weak reference to the primitive *MVar#* itself.

Whereas the implementation of figure 5.3 stops the thread correctly, it leads to problems with GHC's current deadlock detection. Doing *readMVar* on the only direct reference to an *MVar* results in the runtime system assuming that the *MVar* will never be filled. This is a correct assumption without weak references, but in our case there is still a weak reference on the

MVar which can be used to fill the *MVar*. Because of this wrong assumption, the runtime system sends the exception *BlockedIndefinitelyOnMVar* and stops the thread waiting on the *MVar*.

Although we are able to *mask* exceptions and therefore prevent from exceptions being thrown to this thread, this may not be desired in this case, because it would mask all exceptions. There is no way to mask only certain exceptions as desired in our example. Deactivating the deadlock detection in general is also not desired because it provides an otherwise useful feature. Another option would be not to use the *MVar* as the key of the weak reference but an artificial object which would otherwise not be needed. Because it is the *MVar* which has to become unreachable for the thread being unnecessary, we decide not to create an artificial object. Instead we keep the *ThreadId* of the main thread in the thread evaluating the expression reachable until we dereferenced the weak *MVar*. Now this thread may still be able to send asynchronous exceptions to the main thread, so the blocking on the *MVar* is in no case indefinite.

```
type Future a = (MVar a, IORef ())
newFuture :: NFData a ⇒ a → IO (Future a)
newFuture e = do
  m ← newEmptyMVar   – MVar to hold the resulting value
  t ← newEmptyMVar   – MVar to hold the ThreadId
  w ← mkWeakMVar m $ do
    child ← readMVar t
    killThread child
  main ← myThreadId
  child ← forkIO $ do
    n ← e `seq` deRefWeak w
    case n of
      Nothing → return ()
      Just k  → main `seq` putMVar k e
  putMVar t child
  return m
```

Figure 5.4.: Using a weak reference on a dummy variable to stop a thread.

The implementation can be seen in figure 5.4. Its pattern can also be used when multiple results are written into a *Chan* instead of an *MVar* and is used in various implementations for this thesis.

5.2. Bag of Tasks Implementation

After presenting the general idea of a *bag of tasks* in section 4.4.1, this section concentrates on the implementation of this framework. The idea behind the bag of tasks is simple and

comparable to that of *thread pools*. A small number of worker threads execute an often much larger number of tasks.

The Haskell package *threads-pool* of Ilya Portnov[2] implements a similar approach. However, thread pools, including this implementation, are often not designed to support choosing between different kinds of task buffers. Additionally, it is often not possible to add new tasks from another task, which has been used to implement the search strategies on trees. Because of this, we decided to write our own bag of tasks implementation, which is described in this section.

The Low-Level Interface of the Bag of Tasks

The central function provided by the bag of tasks framework is *runTaskBag*.

$$runTaskBag :: BufferType \rightarrow [\,Task\ a\ (Maybe\ a)\,] \rightarrow IO\ [\,a\,]$$

This function is used to start the evaluation of a list of tasks with the given task buffer type and creates a list to be evaluated lazily. Basically *runTaskBag* has to initialise the bag, which includes creating the task buffer and starting the worker threads, add the tasks to the task buffer, and make sure the computation is terminated once the list of results is garbage collected.

In our implementation, *runTaskBag* is just a thin abstraction layer delivering a less complex interface to the basic bag of tasks implementation. The basic implementation provides a set of functions implemented mostly based on the *IO* monad providing the main logic of the bag of tasks. The function *newBag* creates a new empty bag of tasks.

$$newBag :: BufferType \rightarrow IO\ (Bag\ a)$$

It creates the task buffers, some shared variables, and starts the worker threads. The number of these is equivalent to the number of system threads of the GHC runtime system. Using more threads would hardly increase the amount of parallelisation while increasing the synchronisation and context switching overhead. The bag of tasks returned by *newBag* can be used for tasks returning values of only one type, *a*.

Having created a new bag of tasks, it is possible to add tasks with the function *addTask*. The tasks of this implementation level are different from those of the user interface. They are defined in the *IO* monad and, similarly to the other tasks, either return a value or not.

$$addTask :: Bag\ a \rightarrow IO\ (Maybe\ a) \rightarrow IO\ ()$$

Tasks can be added from anywhere, from the outside of the bag or from another task.

$$getResult :: Bag\ a \rightarrow IO\ (Maybe\ a)$$

[2] `http://hackage.haskell.org/package/threads-pool` [accessed 25-May-2014]

These results can be received with the function *getResult*. This function returns a result if there are any results available. If there are no computed results, it either blocks or returns *Nothing*. Whichever happens depends on the current state of the bag of tasks.

Furthermore, it is possible to call *noMoreTasks* to indicate that no more tasks will be added from the outside of the bag.

$$noMoreTasks :: Bag\ a \longrightarrow IO\ ()$$

However, after calling *noMoreTasks*, the worker threads may still add additional tasks during the execution of another task. The decision on whether it is still possible to get further results is easy. It is possible as long as tasks are in the task bag, either being executed by one of the worker threads or waiting in the task buffer.

Furthermore, it may sometimes be required to stop the computation of the bag, when there are still tasks, but the user knows that there will not be any results requested with *getResult*.

$$terminateBag :: Bag\ a \longrightarrow IO\ ()$$

Some search strategies from section 4.4.2 required the possibility to return results in the middle of the execution of a tasks. This can be done with *writeResult*, which writes one result to the output:

$$writeResult :: Bag\ a \longrightarrow a \longrightarrow IO\ ()$$

Based on this interface it is easily possible to define the abstraction layer *newTaskBag*. In contrast to the basic bag of tasks interface, the abstraction layer uses another formulation for tasks. The implementation of these is straight forward and in the end it provides the interface seen in figure 5.5

$$\begin{aligned}
&\textbf{newtype}\ Task\ r\ a\\
&\textbf{instance}\ Monad\quad (Task\ r)\\
&\textbf{instance}\ MonadIO\ (Task\ r)\\
&\textbf{type}\ WriteResult = (r \longrightarrow IO\ ())\\
&\textbf{type}\ AddTask\quad = (IO\ (Maybe\ r) \longrightarrow IO\ ())\\
&runTask\quad :: Task\ r\ (Maybe\ r) \longrightarrow AddTask \longrightarrow WriteResult \longrightarrow IO\ (Maybe\ r)\\
&addTask\quad :: Task\ r\ (Maybe\ r) \longrightarrow Task\ r\ ()\\
&writeResult :: r \longrightarrow Task\ r\ ()
\end{aligned}$$

Figure 5.5.: The task's interface.

The Basic Implementation

The bag of tasks is implemented using *Software Transactional Memory*, which allows modifying the state of the buffer atomically without having to care about locks and their order. The shared state can be seen in figure 5.6. It consist of a channel for the results *resultChan*,

```
data Bag r = Bag {
    resultChan     :: TChan r
  , taskBuffer     :: TaskBuffer (IO (Maybe r))
  , terminateVar   :: TVar Bool
  , moreTasksVar   :: TVar Bool
  , workerStates   :: Map ThreadId (TVar Bool)}
```

Figure 5.6.: The shared state of the bag of tasks.

```
getResult :: MonadIO m ⇒ Bag r → m (Maybe r)
getResult bag = atomically $ do
  result ← tryReadTChan (resultChan bag)
  case result of
    Just r    → return (Just r)
    Nothing → do
      terminated ← readTVar (terminateVar bag)
      unless terminated $ do
        moreTasks ← readTVar (moreTasksVar bag)
        when moreTasks retry
        noTasks ← and `liftM` (mapM isEmptyBufferSTM (bufferList bag))
        unless noTasks retry
        mapM_ (λtvar → readTVar tvar >≈ (flip when) retry)
          (Map.elems (workerStates bag))
      return Nothing
```

Figure 5.7.: The shared state of the bag of tasks.

the buffer for waiting tasks *taskBuffer*, a flag to indicate that the computation has been terminated *terminateVar*, a flag to indicate if there will be more tasks added to the task buffer *moreTasksVar*, and the states of the worker threads. A state of the worker thread shows if the corresponding thread is working on a task (*True*) or if it is idling (*False*).

How do we know that there are no values left? In STM transactions, we can check if there are values in the *resultChan*. If this is not the case, we should first check, whether the bag has been terminated or whether we got the information that there are no more tasks being added to the bag indicated by *moreTasksVar*. In case it is not possible that there will be new tasks, we have to inspect the *workerStates* and the *taskBuffer*. If none of the workers is working on the task and if there are no tasks in the *taskBuffer*, there will be no values coming from the bag. Then, *getResult* can safely return *Nothing*. The code of *getResult* can be seen in figure 5.7.

```
data Result a = NoResult | Result a | NoMoreTasks
data Bag r = Bag {
  workers           :: [ThreadId]
  , taskBuffer      :: TaskBuffer (IO (Maybe r))
  , resultChan      :: Chan (Result r)
  , taskCountVar    :: MVar Int
  , moreTasksVar    :: MVar Bool
  , waitingOnMoreTasksVar :: MVar Int
  }
```

Figure 5.8.: The shared state of the bag of tasks in the variant without using STM.

Alternative Implementation without Using STM

On one hand, since STM is a rather high level interface, it may be expected to have a certain overhead compared to using locks manually. On the other hand STM is implemented very efficiently and allows writing down complex dependencies quickly.

This section defines an alternative implementation of the bag of tasks without using STM to compare the runtimes with the first implementation.

The interface of this is the same as the basic interface presented above. The shared state of the bag of tasks can be seen in figure 5.8. In contrast to the implementation above, this does not contain a worker state. Instead of this, it stores the number of tasks in the bag with *taskCountVar*.

How do we know that there are no values left? On adding a new task in the task buffer the *threadCount* is increased. This has to be done before writing an intermediate result into the *resultChan*. The number of tasks stored in *taskCountVar* is decreased in *getResult* only. Before reading a value from the channel it asks if more values may be added later.

```
moreTasks ← readMVar (moreTasksVar bag)
```

If so, it can assume that there will be a value available later for now. If *moreTasksVar* is set to *False* there will also be a message through the result channel and *getResult* is restarted once the message has been put back for other threads which wait on this variable indicated by *waitingOnMoreTasksVar*. If *moreTasks* is *False*, we just have to read the *taskCountVar* to find out if there will be values left.

Multiple Task Buffers

Until now, *runTaskBag* was said to create only one common buffer of waiting tasks, but this is not the only option available. It is also possible to use one task buffer per thread, which

reduces the synchronisation overhead between the threads. When a task buffer of one thread is empty, it queries the task buffers of the other threads to hand over one or multiple tasks. To allow the interaction with a foreign task buffer without involvement of the other task, these buffers are also accessed via STM actions.

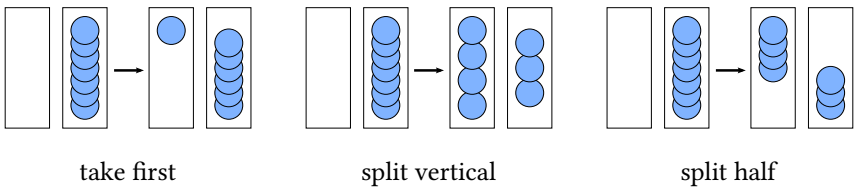


Figure 5.9.: Visualisation of the splitting strategies.

There are multiple possibilities of how to decide which tasks from the foreign buffer are taken. Figure 5.9 shows all strategies implemented here. One way is to just take the first task in the foreign buffer. We call this method *take first*.

Following the naming of Vieira et al. [32], the other strategies are called *half splitting* and *vertical splitting*. Both strategies divide the task buffer of one thread in halves. This takes more time, but it will limit the number of splits.

With vertical splitting, the task buffer is divided alternately. As a result, the splitting function has to read the complete buffer. This is also the case for half splitting. Here, the own worker takes the first half of the task buffer whereas the foreign worker continues to evaluate the second half.

6. Evaluation

This chapter is about the evaluation of certain important criteria of parallel search strategies. This includes besides the performance (section 6.3) the desired degree of completeness (section 6.1) and the behaviour in case of an abortion of the computation (section 6.2). These criteria are verified with the help of the test programs presented in the following sections.

6.1. Completeness

We described both search strategies being complete for deterministic loops and search strategies which are only complete for infinite non-deterministic search trees.

Search strategies competing with breadth-first search have to be complete for non-deterministic loops. The strategies *bfsParallel*1, *bfsParallel*2, *bfsParallel*3, *bfsBag*, *fairSearch*1, *fairSearch*2, and *fairSearch*3 fulfil this requirement. To verify this, we write a function which non-deterministically yields all natural numbers beginning at n.

$$f\ n = f\ (n + 1)\ ?\ n$$

The call $f\ 0$ =:= 1 should then yield the value *success*. However, the search tree of this call is infinite. It can be seen at the left in figure 6.1. A normal depth-first search on this tree would not find any *success* in this tree.

We could argue that there is also an incomplete strategy finding this result by searching the right subtree first. Therefore, we also define g, which is semantically equivalent to f, but results in a different search tree seen at the right side in figure 6.1:

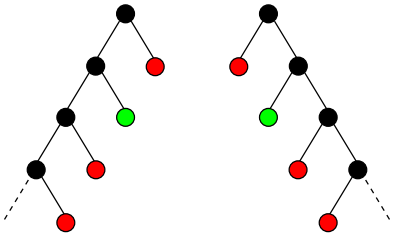


Figure 6.1.: The search tree of $f\ 0$ =:= 1 (left) and the search tree of $g\ 0$ =:= 1 (right).

```
g n = n ? g (n + 1)
```

Incomplete strategies would diverge on trying to get the first result of $f\ 0 =:= 1$ or $g\ 0 =:= 1$. Strategies fulfilling this completeness criterion return the first result *success* in both cases similar to all strategies mentioned above. This test has to be performed using one thread only; otherwise the strategies using bag of tasks would start two threads simultaneously and only one of these would start evaluating the infinite branch.

Even more complete than breadth-first search are fair search strategies from section 4.5. These have to return an existing result even if there are deterministic loops in the search tree. A minimal example would be the goal *loop* ? () which obviously contains the result (). Fortunately, simple loops are detected automatically by the GHC runtime system, so *loop* ? () is not a valid test. As a foundation for our test, we define the function *fib* which returns the Fibonacci number with the given index:

```
fib n = case n of
  0 → 0
  1 → 1
  n → fib (n - 1) + fib (n - 2)
```

Furthermore we define the list of all Fibonacci numbers *fibs*:

```
fibs = fibs' 0
  where
    fibs' n = fib n : fibs' (n + 1)
```

Because the list of all Fibonacci numbers is infinite, the evaluation of *last fibs*, the last element of this list, diverges. We use the following goals to test our fair strategies for completeness:

```
last (fibs ? [1])
last ([1] ? fibs)
```

All fair strategies, *fairSearch*1, *fairSearch*2, and *fairSearch*3, return the existing result 1 while breadth-first search diverges at the evaluation of the first result of *last* (*fibs* ? [1]). Again, this test has to be evaluated using one system thread only.

6.2. Abortion Behaviour

Often, complete Curry programs perform one or multiple searches during their runtime and continue with the processing or output of the values. When requesting only one value with *getOneValue*, the search does not have to be completed. However, parallel search strategies often create sparks or start threads to search parts of the search tree in parallel. Ideally, this computation should be stopped once the first values have been returned. To ensure this, the bag of tasks implementation and the fair search strategies implement the termination of the threads with a technique presented in section 5.1.

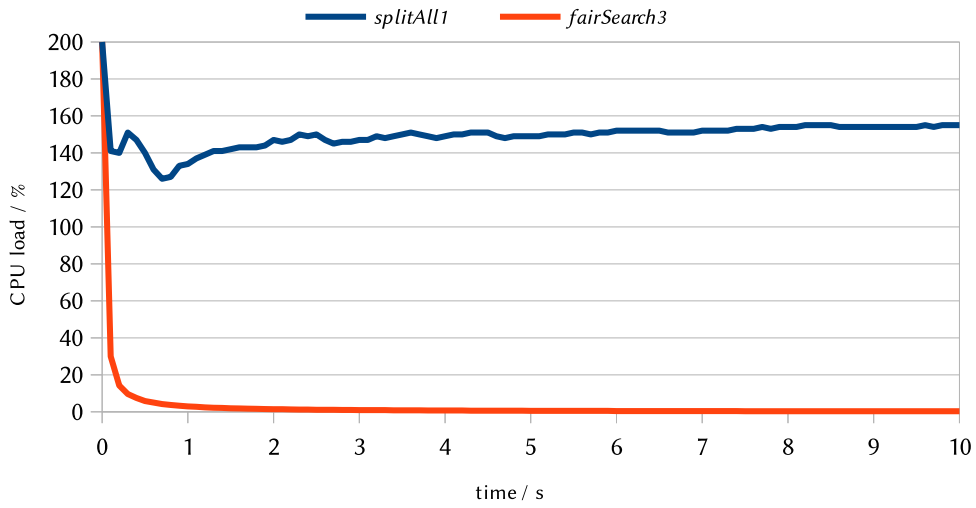


Figure 6.2.: CPU load during the execution of the test program for two example strategies *splitAll*1 and *fairSearch*3.

To check the behaviour of the search strategies in this case, we define a test program based on the goal *last* ([1] ? *fibs*) from section 6.1:

```
main strategy = do
  Just x ← getOneValue strategy (last ([1] ? fibs))
  putStrLn ("Got the value " ++ show x)
  sleep 10
```

sleep n is an action which suspends the evaluation of the Curry program for *n* seconds. As the search strategies except *splitLeft* and *dfsBagLeft* evaluate the left subtree before the one at the right, all strategies return the result 1 first, but they differ in terms of CPU load during the phase of sleeping.

The different behaviour after getting the first value can be seen in figure 6.2. This diagram compares the CPU load during the evaluation of *main* for the search strategies *splitAll*1 and *fairSearch*3. *fairSearch*3 represents search strategies correctly terminating the search in the rest of the search tree whereas *splitAll*1 is a strategy that does not terminate the computation of the last Fibonacci number. Unfortunately, the behaviour of *splitAll*1 depends on the implementation of sparks in the GHC runtime system and we have no direct influence on this. Table 6.1 shows that all strategies implemented explicitly with threads fulfil this requirement whereas the strategies implemented using sparks continue the computation.

terminates correctly	continues with evaluation
fairSearch1	parSearch
fairSearch2	splitAll1
fairSearch3	splitAll2
dfsBag	splitAll3
fdfsBag	splitLimitDepth
bfsBag	splitAlternating
dfsBagCon	splitExponential
fdfsBagCon	splitRight
bfsBagCon	splitLeft
dfsBagLimit	bfsParallel1
dfsBagRight	bfsParallel2
dfsBagLeft	bfsParallel3

Table 6.1.: The abortion behaviour of all presented search strategies.

6.3. Performance Analysis

An important goal of this thesis is to improve the performance with parallel evaluation. In order to decide for or against a certain search strategy it is necessary to measure and compare this performance. While some strategies improve the evaluation speed, some offer a higher level of completeness compared to existing search strategies (section 4.5). The goal for these is not to reduce the performance too much, so these strategies can still be used to evaluate most programs.

6.3.1. Benchmark Programs

To measure the performance of the developed strategies, we select a set of benchmark programs. Among these are traditional logic programming examples such as *SearchQueens* and *PermSort* and examples from the praxis like *EditSeq*. Other benchmarks require a certain level of completeness (section 3.2) like *NDNums*, so they cannot be evaluated with every search strategy.

n Queens Puzzle

The *8 queens puzzle* is the problem of placing 8 chess queens on a chess board, without any queen being able to capture another. The generalisation of this problem is the n queens puzzle, in which we have to place n queens on a chess board of $n \times n$ squares. As a queen can move any number of squares on the same rank, file, or diagonal, we cannot place two queens in the same row, column, or diagonal.

To represent possible solutions, we index both rows and columns with numbers from 1 to n. As two queens cannot be placed in the same column, we can represent the positions of all queens as the list of rows of the occupied squares. Using this notation, $[1, 2, 3, 4, 5, 6, 7, 8]$ would mean that eight queens stand on the diagonal from the top left corner to the bottom right corner of an 8×8 board. As two queens are not allowed to occupy a square in the same row, in a safe placing, each number only appears once in its representation. Therefore, the safe placements are among the permutations of the list $[1\,..\,n]$.

The computation of the safe positions is implemented non-deterministically using the *generate and test* principle in figure 6.3. The benchmark is called *SearchQueens*. The operation *queens* implements this search and thus enumerates all possible safe placements with *permute*. The placement is safe if and only if there is no pair of queens standing in the same diagonal. This is expressed by *allSafe* with the help of the operation *safe*.

Note that the implementation of *insert* is not the naïve implementation, which forces the evaluation of the given list to weak head normal form, before deciding on where to put the value x.

$$\begin{aligned} &\mathit{insert}\ x\ [\,] &&= [x] \\ &\mathit{insert}\ x\ (z : zs) &&= x : z : xs\ ?\ z : (\mathit{insert}\ x\ zs) \end{aligned}$$

We consider the first rule to be equal to the left side of the choice of the second rule. Our implementation seen in figure 6.3 results in the search tree visible in figure 6.4. This shows the search tree of the list permutations. At certain points, a beginning of a list is written at a branch. This is the list which is already evaluated at this point of the search tree. All lists below start with these values. The red parts of the tree are parts that do not have to be evaluated, because the program knows from the already determined beginning of the list, that this placement will not be safe. This is detected by the deterministic computation in *allSafe* which happens at each point where a new value comes to the prefix of the list.

Permutation Sort

Another classical example for logic programming is *permutation sort*. Permutation sort is in no way an efficient algorithm, but its implementation is very close to the typical specification of sorting functions: a sorting function returns a sorted permutation of the given list.

Permutation sort is defined with the help of the operation *sorted*, which takes a list and returns the same list only if it is sorted. If the given list is not sorted, the operation *fails*.

$$\begin{aligned} &\mathit{sorted} :: [\mathit{Int}] \longrightarrow [\mathit{Int}] \\ &\mathit{sorted}\ [\,] = [\,] \\ &\mathit{sorted}\ [x] = [x] \\ &\mathit{sorted}\ (x : y : ys) \mid x \leq y = x : \mathit{sorted}\ (y : ys) \end{aligned}$$

With this simple operation, *sort* can be defined as:

```
queens :: Int → [Int]
queens n | allSafe qs = qs where qs = permute [1 . . n]

insert :: a → [a] → [a]
insert x xs = x : xs
            ? case xs of
                z : zs → z : insert x zs

permute :: [a] → [a]
permute [ ]      = [ ]
permute (x : xs) = insert x (permute xs)

allSafe :: [Int] → Bool
allSafe qs = allSafe' (zip qs [1 . .])
  where
    allSafe' :: [(Int, Int)] → Bool
    allSafe' [ ]          = True
    allSafe' (xy : xys) = all (safe xy) xys ∧ allSafe' xys

safe :: (Int, Int) → (Int, Int) → Bool
safe (a, b) (c, d) = abs (a − c) ≠ abs (b − d)

abs :: Int → Int
abs x | x < 0      = −x
      | otherwise = x

main = queens 8
```

Figure 6.3.: The implementation of *SearchQueens*.

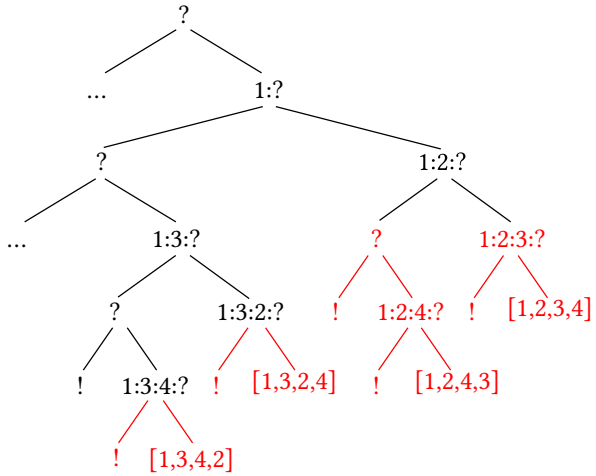


Figure 6.4.: A part of the search tree of *queens* 4.

$$sort\ xs = sorted\ (permute\ xs)$$

As permutation sort uses the same permutation operation as *SearchQueens*, its search tree is very similar.

Editing Sequences

Comparing genomes is a popular problem in bioinformatics. As genomes can be expressed sequentially, comparing genomes is equivalent to comparing sequences of letters. Letters are represented by constructors of a special data type.

$$\textbf{data}\ Letter = A \mid C \mid G \mid T \mid I \mid R \mid N \mid E \mid D \mid L$$

Example string can then look like the following.

$$\begin{aligned} airline &= [A, I, R, L, I, N, E] \\ darling &= [D, A, R, L, I, N, G] \end{aligned}$$

Comparing sequences can be done by editing one sequence to match another. This editing can be a replacement, a deletion, or an insertion, and may be followed by further editing operations.

$$\begin{aligned} \textbf{data}\ Edit = {} & Replace\ Letter\ Letter\ Edit \\ \mid {} & Delete\ \ Letter\ \ \ \ \ \ Edit \\ \mid {} & Insert\ \ Letter\ \ \ \ \ \ Edit \\ \mid {} & Empty \end{aligned}$$

Not changing anything at the current position is also counted as a replacement like *Replace A A*.

Often, we are interested in editing operations with a minimal number of changes and sometimes we may only be interested in the minimum number of changes. Thus, we specify the number of changes as *value*.

$$\begin{aligned} &value :: Edit \longrightarrow Int \\ &value\ (Insert\ \_\ x) = 1 + value\ x \\ &value\ (Delete\ \_\ x) = 1 + value\ x \\ &value\ (Replace\ a\ b\ x) = value\ x + \textbf{if}\ a \equiv b\ \textbf{then}\ 0\ \textbf{else}\ 1 \\ &value\ Empty = 0 \end{aligned}$$

Both insertion and deletion are counted as a *value* of 1. A replacement with the same letter has the *value* of 0 and a replacement with a different letter also has the *value* of 1.

The main goal is defined in terms of the *generate and test* paradigm. The generator *editMax* generates edit operations with a maximum length. *simpLR e*=:=(*airline*, *darling*) tests whether the editing operation results in the correct sequences.

$$\begin{aligned} &goal \mid simpLR\ e =:= (airline, darling) \\ &\quad = (value\ e, e) \\ &\ \ \textbf{where}\ e = editMax\ 14 \end{aligned}$$

In addition to the edit operation itself, *goal* returns the *value* of the edit operation.

Then, the function *simpLR* has to calculate both the source and the goal sequence from the editing operations.

```
simpLR :: Edit → ([Letter], [Letter])
simpLR (Replace a b x) = (a : x1, b : x2) where (x1, x2) = simpLR x
simpLR (Delete a x)    = (a : x1,     x2) where (x1, x2) = simpLR x
simpLR (Insert  a x)   = (    x1, a : x2) where (x1, x2) = simpLR x
simpLR Empty           = (    [],    [])
```

The replace operation *Replace* adds a letter to each sequence whereas both the delete operation *Delete* and the insert operation *Insert* only add their argument letter to the left respectively the right sequence.

editMax creates the sequences then. The given argument is the sum of the length of both result sequences.

```
editMax m =
  if m ≤ 0
    then Empty
    else Replace _ _ (editMax (m − 2))
      ? Delete   _   (editMax (m − 1))
      ? Insert   _   (editMax (m − 1))
      ? Empty
```

To properly compare two sequences, it is necessary to find the shortest of all editing operations. Thus, we have to compare the first element of all results of *goal*. The function *main* calculates all these values lazily with the given search strategy. After calculating the minimum of the values, all minimal editing operations are written to the output.

```
main strategy = do
  vlist ← getLazyValues strategy goal
  let m = minlist (map fst vlist)
  putStrLn ("Minimum: " ++ show m)
  putStrLn (unlines (map show (filter (λx → fst x ≡ m) vlist)))
```

This program is far from being optimal. Due to generating all editing operations up to a given length limit, a large search space is created. While the size of this search tree is reduced by the narrowing implementation its size is still vast. Additionally, the number of results is also comparingly large. The example goal comparing *airline* and *darling* yields 48,639 results. Benchmarking this shows the effort at parallelising unoptimised Curry programs, which have been written to mostly resemble the problems' definition.

Half

The program *half* divides a given number by two using guessing:

$$half\ y \mid equal\ (add\ x\ x)\ y = x\ \textbf{where}\ x\ \textbf{free}$$

The numbers are thereby defined as peano numbers:

$$\textbf{data}\ Peano = Zero \mid Successor\ Peano$$

The functions *add* and *equal* are implemented as expected. In addition, to convert a normal number of type *Int* to a peano number, we define the function *toPeano*. The main goal then compares the given number 800 with the half of another number 1600, which is obviously true.

$$main = equal\ (toPeano\ 800)\ (half\ (toPeano\ 1600))$$

Last

The program last is the simplest of all benchmark programs used here. It does a search for the last element in a list using free variables.

$$last :: [a] \to a$$
$$last\ l \mid xs +\!\!+ [x] =:= l = x\ \textbf{where}\ xs, x\ \textbf{free}$$

The above code can be read as: when the argument is equal to any list xs with the appendix $[x]$, then x is the last element.

The search tree, which can be seen for an example list in figure 6.5, results from the implementation of narrowing in Curry. It has nearly the form of a list while it has failures alternating at each side. For the benchmarks, we use the main goal *last* (*replicate* 10000 *True*).

NDNums

NDNums is our only benchmark program creating an infinite search tree. From section 6.1, we already know the function yielding all integers starting at the integer provided as an argument:

$$f\ n = f\ (n + 1)\ ?\ n$$

The search tree created by f is infinite but very narrow. Each level in this search tree contains only two tree nodes. The function h is semantically equivalent to f, but it has a much wider search tree:

$$h\ n = h\ (n + 1)\ ?\ n\ ?\ h\ (n + 1)$$

The main goal compares the result of h with 19.

$$main \mid h\ 0 \equiv 19 = success$$

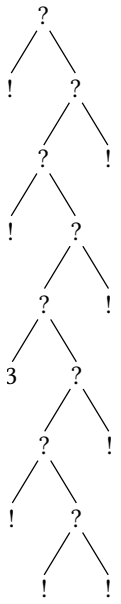


Figure 6.5.: The Search Tree resulting from the goal *last* $[1, 2, 3]$.

Similar to *Last*, the amount of computation per node is very low here since it only contains adding two numbers or comparing them. However, this benchmark tests the breadth-first and fair search strategies on how they handle wide search trees.

6.3.2. Benchmarking System

All benchmarks were run on a system with two *Intel Xeon CPU E5-2620* and 16 GiB of RAM using Debian GNU/Linux 7.3 with Linux Kernel 3.2.51. The Intel Xeon E5-2620 has been introduced in the first quarter of 2012 and each has six processing cores. All of these cores support Hyper-Threading Technology; therefore each physical core appears as two logical cores. All in all, the system has 12 physical and 24 logical cores.

To compile the generated Haskell code, we used a development snapshot of the Glasgow Haskell Compiler 7.8 from January 9, 2014, because Glasgow Haskell Compiler 7.6 does not feature a thread-safe version of the unique identifier supply.

6.3.3. Results

This section shows the results of the benchmarks by comparing similar strategies with each other. Earlier benchmarks showed that, among the id supplies currently delivered with KiCS2,

the default supply *ioref* has the best performance. Therefore, this id supply has been used for all benchmarks in this section.

As the runtime and the memory consumption of a program depends on the scheduling by the operating system and by the GHC runtime system, we ran each benchmark multiple times. The shown values are the mean of these multiple runs. We ran each benchmark at least four times. The benchmarks computing only one value in the search tree have been run eleven times, since the scheduling has more influence on the evaluation time here.

Deterministic Parallel Depth-First Strategies

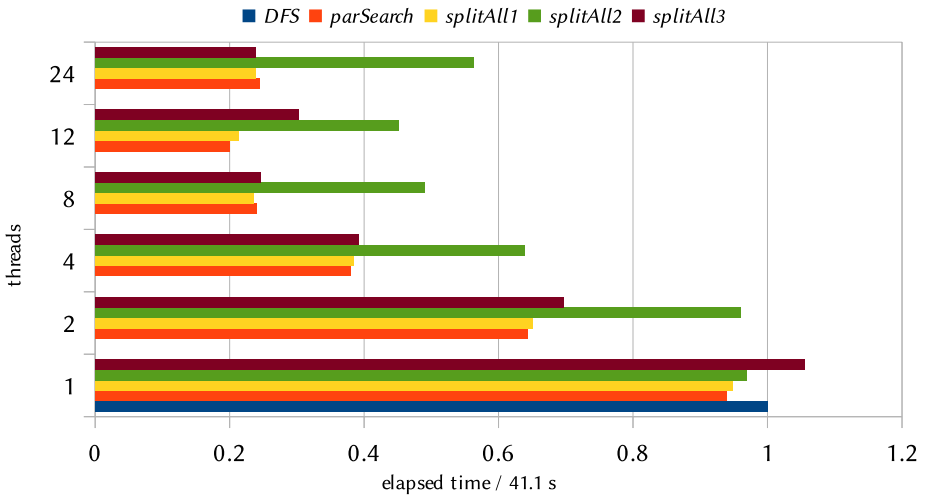


Figure 6.6.: Normalised diagram of the elapsed time on computing *one* solution of the *PermSort* benchmark. Comparing all parallel deterministic depth-first search strategies.

Section 4.3 defines four parallel deterministic depth-first search strategies. The diagrams in figure 6.6, 6.8, and 6.7 compare these strategies in terms of runtime of different benchmark programs. All of these diagrams show, that *parSearch* (red) and *splitAll*1 (yellow) are on par with each other. We expected this, because *splitAll*1 was meant to be a formulation variant of *parSearch* using the *Eval* monad instead of the simple function *par*.

*splitAll*2 implements the approach of separating the parallel evaluation strategy from the traversal of the tree. As expected, this is slower compared to directly traversing the tree in parallel. Though, in *PermSort One* (figure 6.6), it still reaches a runtime of 18.57*s* at 12 cores in comparison to the 41.1*s* of the sequential depth-first search .

Between the strategies *splitAll*1 (yellow) and *splitAll*3 (brown) it is not that clear which one is the winner. *splitAll*3 wins in benchmarks yielding many values like *EditSeq* (figure 6.7)

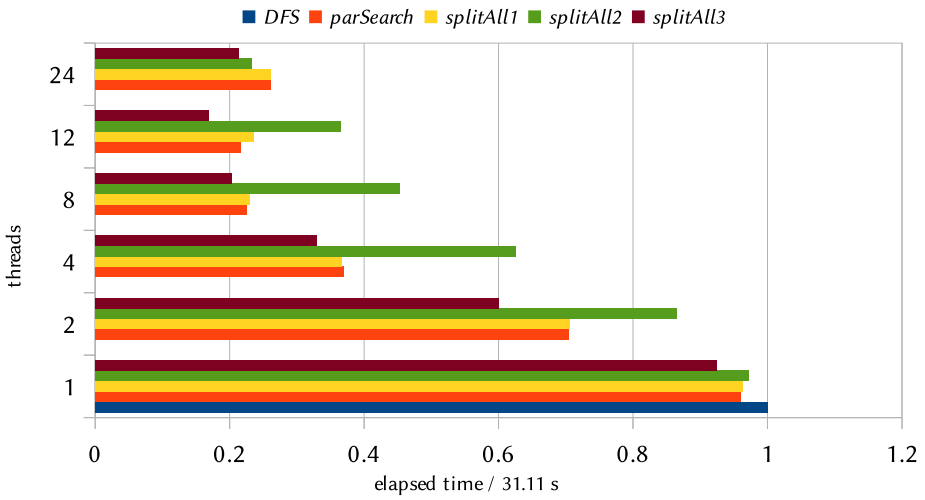


Figure 6.7.: Normalised diagram of the elapsed time on running the *EditSeq* benchmark. Comparing all parallel deterministic depth-first search strategies.

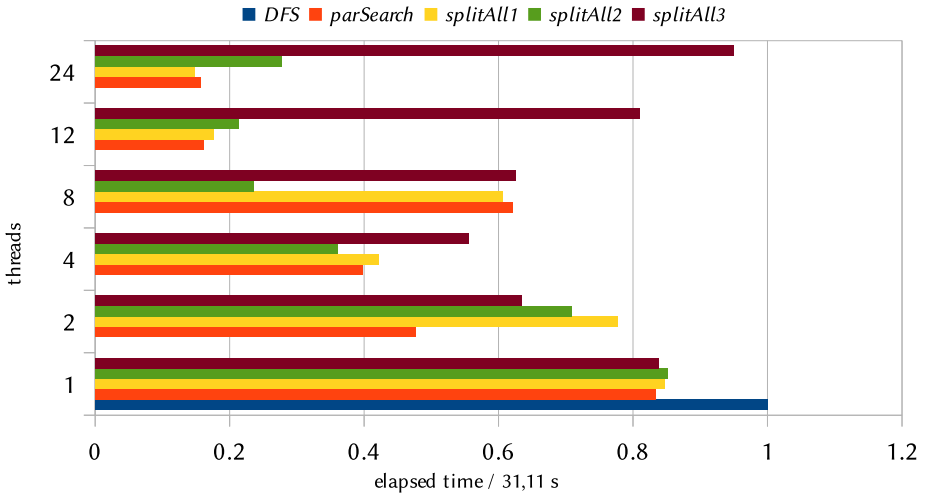


Figure 6.8.: Normalised diagram of the elapsed time on computing *all* solutions of the *Half* benchmark. Comparing all parallel deterministic depth-first search strategies.

whereas *splitAll*1 wins in benchmarks yielding only few values like *PermSort* (figure 6.6). However, a huge difference between both can be seen in figure 6.8. The runtimes of *splitAll*3 are often even worse than those of *splitAll*2.

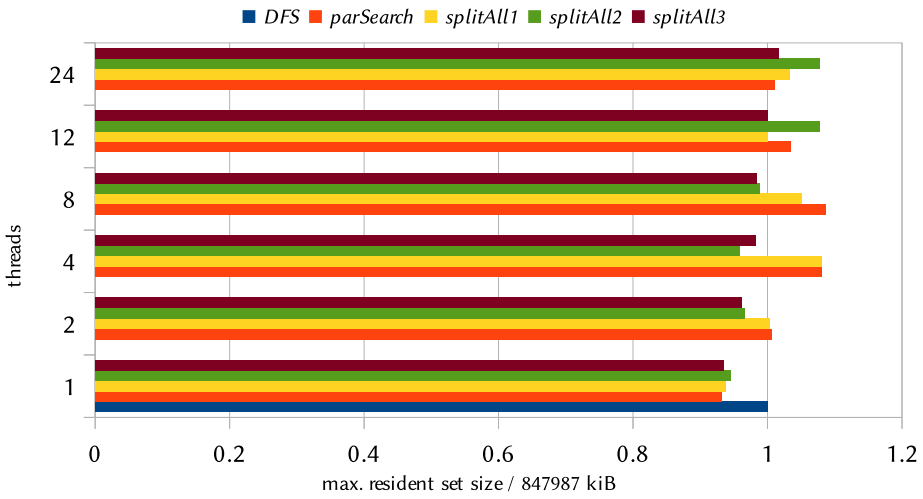


Figure 6.9.: Normalised diagram of the maximum used memory during the computation of the *EditSeq* benchmark. Comparing all parallel deterministic depth-first search strategies.

Figure 6.9 shows the memory usage of the search strategies compared in this section. All strategies do not differ significantly from the sequential depth-first search.

Figure 6.10 shows an example in which all parallel deterministic search strategies perform worse than the sequential depth-first search. This is due to the list-like search tree of *Last* with a very little amount of sequential evaluation.

Interestingly, except from *PermSort*, no benchmark profits from this parallelisation strategies on the computation of only one value. The explanation is simple: if a left subtree already contains a value, there is no use from having sparked the right subtree. If the left subtree does not contain a value, the value from the right subtree can not be returned before the left subtree has been evaluated completely. This initially led us to the definition of the bag of tasks search strategies.

After an overall comparison, *splitAll*1 performs best. In many benchmarks like *EditSeq* and *PermSort* it quarters the runtime. In *EditSeq* from 31.11*s* to 7.15*s* with 8 threads and in *PermSort* from 41.1*s* to 8.79*s* with 12 threads. As a result, we choose *splitAll*1 as the recommended parallel deterministic depth-first search strategy.

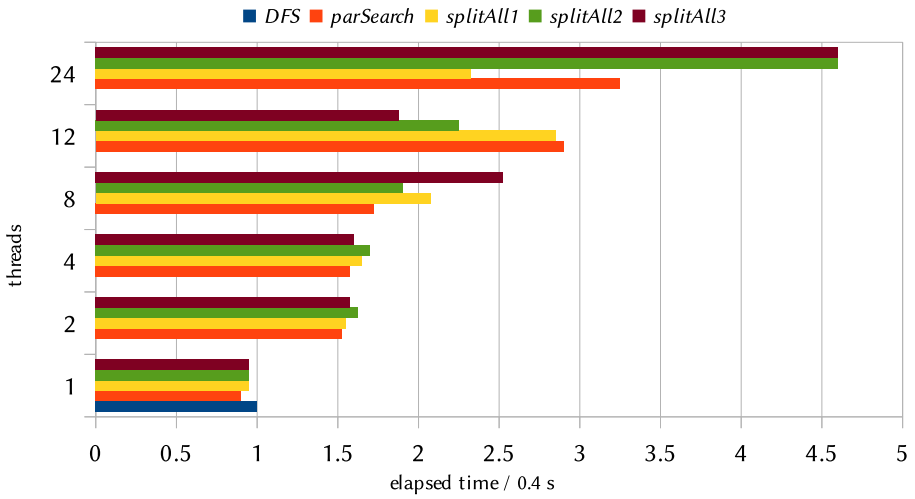


Figure 6.10.: Normalised diagram of the elapsed time on computing *all* solutions of the *Last* benchmark. Comparing all parallel deterministic depth-first search strategies.

Deterministic Parallel Breadth-First Strategies

After having compared the performance of the deterministic parallel depth-first search strategies, this section compares the performance of the breadth-first strategies. *bfsParallel*1 parallelises the evaluation of all nodes in one level whereas both *bfsParallel*2 and *bfsParallel*3 traverse the tree sequentially in breadth-first order while the sparks evaluate the tree in a depth-first order. In *bfsParallel*3 each spark follows a branch down until it reaches a leaf whereas *bfsParallel*2 creates new sparks for each level.

The figures 6.11, 6.12, and 6.13 show the evaluation times for various benchmark programs. First of all, *bfsParallel*1 is never faster than the sequential breadth-first search. Therefore, it is abandoned. As for the parallel depth-first search strategies, no parallel breadth-first search beats the sequential search strategy in the benchmark *Last* (figure 6.12).

*bfsParallel*2 and *bfsParallel*3 produce different results depending on the benchmark programs. For some benchmarks like computing one value of the *PermSort* example, *bfsParallel*3 takes 18.69*s* whereas *bfsParallel*2 takes 32.14*s* with 12 threads. Thereby both is faster than the 70.04*s* needed by the sequential search. The behaviour on computing all values for *PermSort* is similar. For other benchmarks, like computing all values of the *Half* benchmark, *bfsParallel*2 is faster than *bfsParallel*3. *bfsParallel*2 speeds up the computation from 1.51*s* to 0.48*s* with 12 threads and *bfsParallel*3 needs more time than the sequential strategy. Here, the behaviour on computing only one value is also similar.

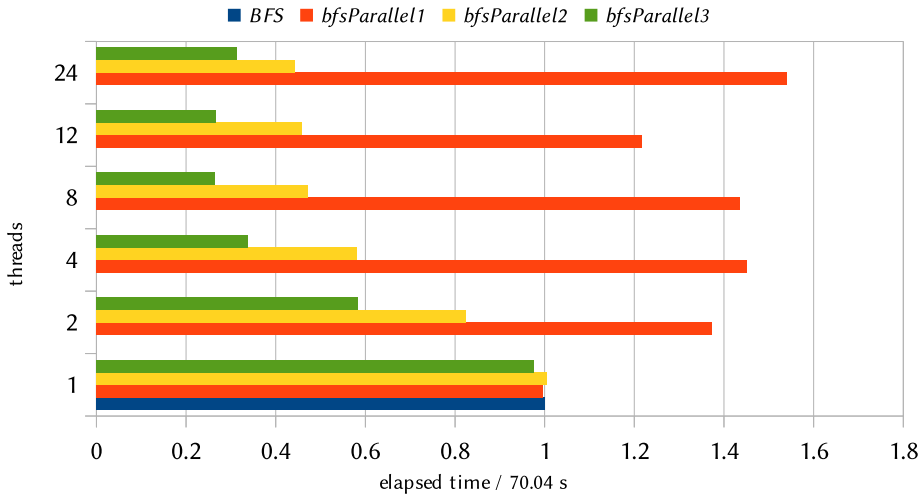


Figure 6.11.: Normalised diagram of the elapsed time on computing *one* solution of the *PermSort* benchmark. Comparing all parallel deterministic breadth-first search strategies.

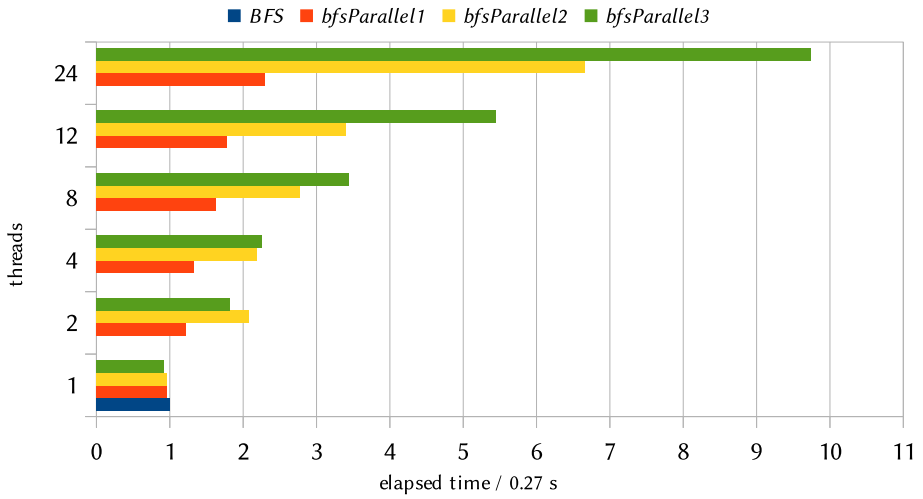


Figure 6.12.: Normalised diagram of the elapsed time on computing *one* solution of the *Last* benchmark. Comparing all parallel deterministic breadth-first search strategies.

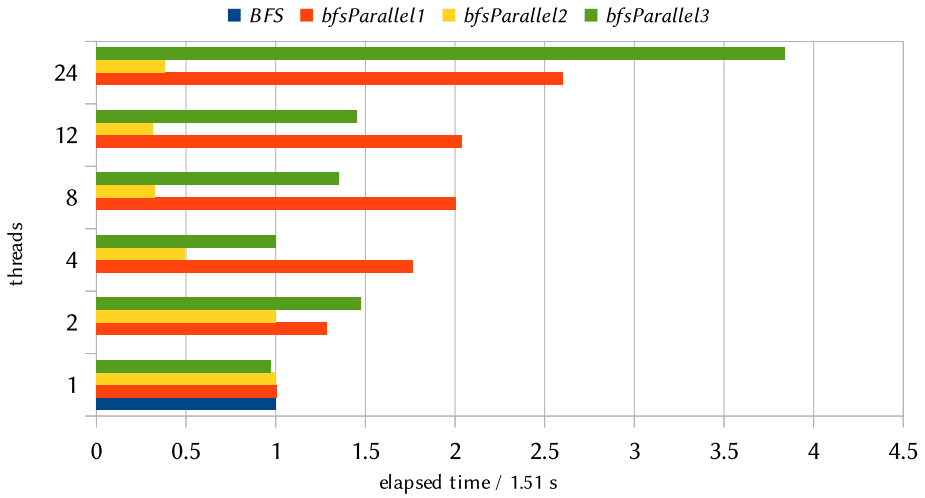


Figure 6.13.: Normalised diagram of the elapsed time on computing *all* solutions of the *Half* benchmark. Comparing all parallel deterministic breadth-first search strategies.

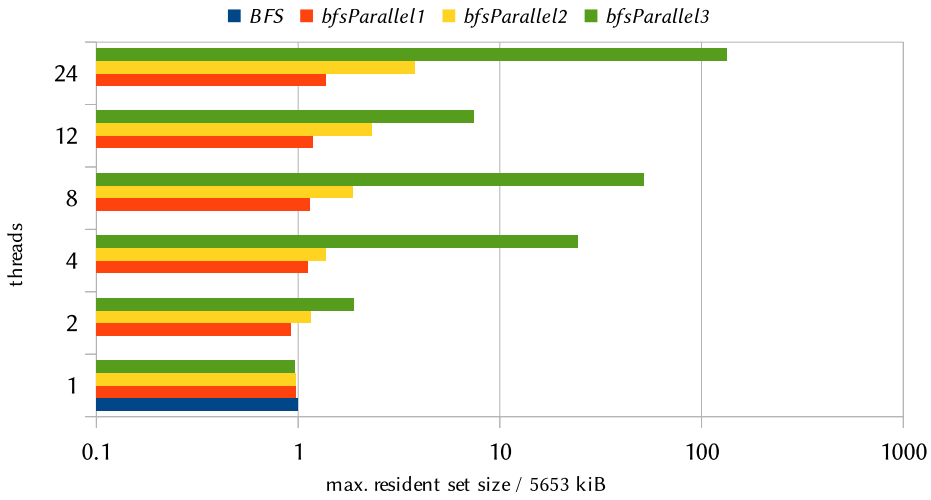


Figure 6.14.: Normalised diagram of the maximum used memory while computing *all* solutions of the *Half* benchmark. Comparing all parallel deterministic breadth-first search strategies.

In terms of memory usage *bfsParallel*3 is often worse than *bfsParallel*2 like in the benchmark of figure 6.14. When looking at this figure, you have to keep in mind the logarithmic scale at the bottom axis. For these reasons, we choose *bfsParallel*2 as the recommended strategy.

Bag of Tasks Search Strategies with One and Multiple Task Buffers

After comparing the performance of multiple deterministic parallel search strategies implemented with semi-explicit parallelism, we now compare the search strategies based on the bag of tasks approach. For the bag of tasks approach, we have three different search strategies: *dfsBag*, *fdfsBag*, and *bfsBag*. Among these, only *bfsBag* reaches the completeness of breadth-first search.

Additionally, we have multiple implementations of the bag of tasks framework. One of these uses a *common buffer* for all worker threads and the others use one task buffer per thread. As long as these have tasks in their own buffer, these implementations do exactly the same. They differ in the behaviour once their own task buffer is empty. The *take first* variant just takes the first of the tasks from a foreign task buffer while *split vertical* and *split half* split up the task buffer in two halves.

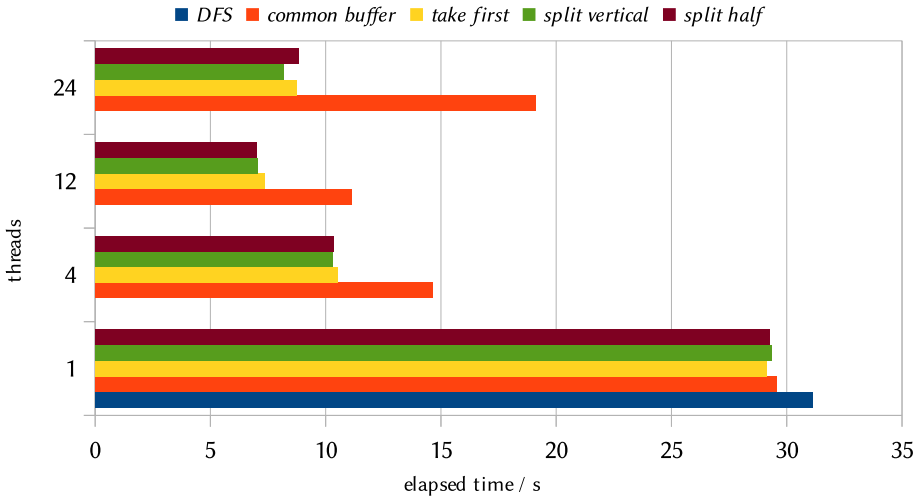


Figure 6.15.: Diagram of the runtime of the *EditSeq* benchmark. Comparing depth-first search strategies based on the bag of tasks approach.

Here, we compare the different implementations of the bag of tasks with each other while using the same search strategy. Figure 6.15 shows that, independent on which bag of tasks implementation used, *dfsBag* reaches resonable speedups compared to the sequential depth-first

search. However, the variant with a common buffer has obviously a higher synchronisation overhead than the other variants. *dfsBag takeFirst* runs the *EditSeq* benchmark in only 7.37*s* compared to 31.11*s* needed by the sequential strategy. All strategies with multiple task buffers are approximately at the same level.

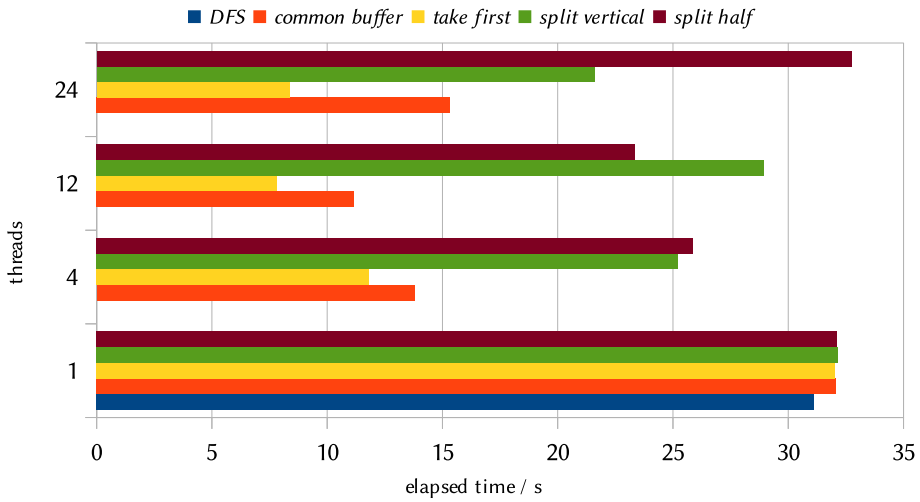


Figure 6.16.: Diagram of the runtime of the *EditSeq* benchmark. Comparing wrong depth-first search (*fdfsBag*) strategies based on the bag of tasks approach.

On the contrary, with both *fdfsBag* (figure 6.16) and *bfsBag* (figure 6.17) both splitting strategies can not beat the *takeFirst* implementation. *fdfsBag* is not slower compared to *dfsBag*, but it often uses more memory.

Other benchmarks like the *Half* benchmark (figure 6.18) show no significant difference between the various bag of tasks implementations, even for *bfsBag*. Like with all search strategies from this thesis, the *Last* benchmark (figure 6.19) does not profit from parallelisation.

For all strategies with all bag of tasks implementation the memory consumption stays within the same range with the sequential implementation. All in all, we decide to use the *take first* implementation of the bag of tasks, as it produces best results with all bag of tasks search strategies. As the *fdfsBag* strategy is never better compared to *dfsBag* we choose to prefer *dfsBag*.

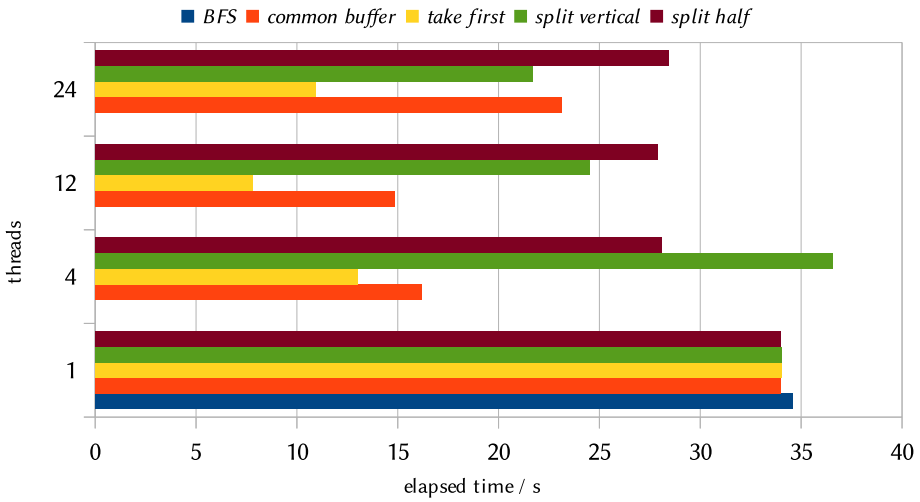


Figure 6.17.: Diagram of the runtime of the *EditSeq* benchmark. Comparing breadth-first search strategies based on the bag of tasks approach.

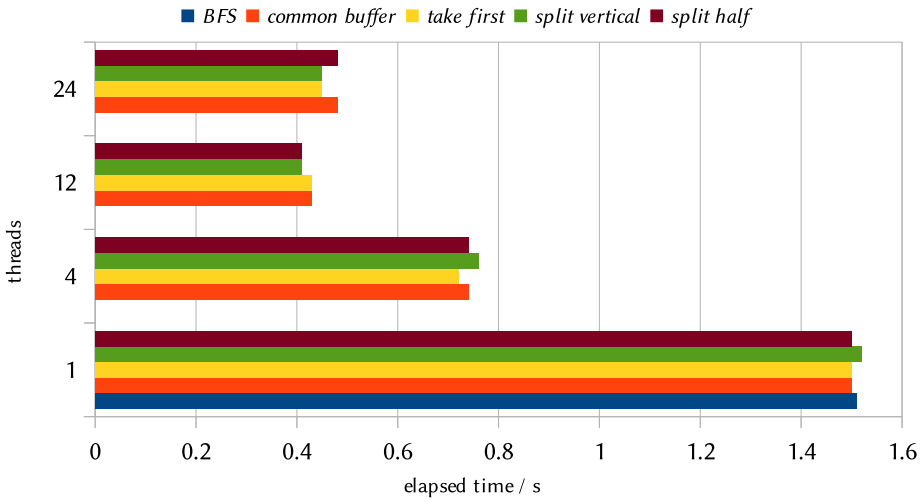


Figure 6.18.: Diagram of the runtime to compute *all* solutions of the *Half* benchmark. Comparing breadth-first search strategies based on the bag of tasks approach.

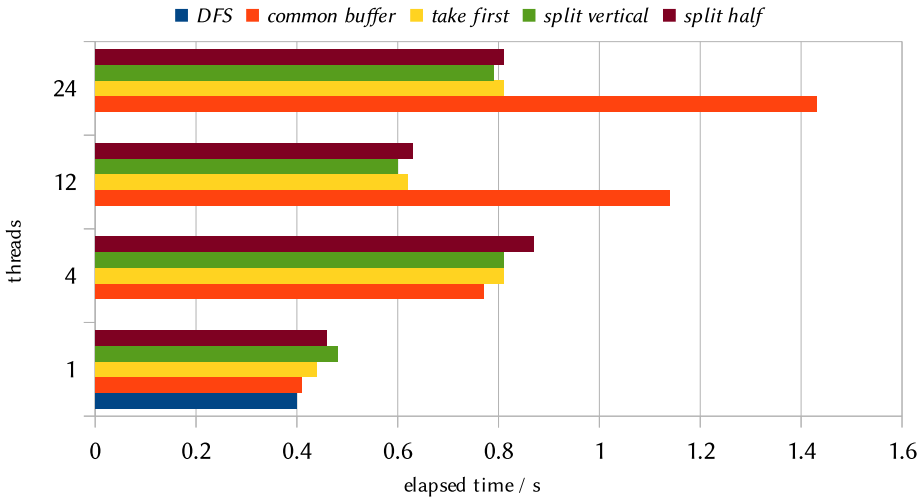


Figure 6.19.: Diagram of the runtime to compute *all* solutions of the *Last* benchmark. Comparing depth-first search strategies based on the bag of tasks approach.

Comparing Non-Deterministic Bag Search Using STM with Bag Search without STM

Even though the bag of tasks search strategies using STM already produce good parallelisation results for depth-first and breadth-first search, we try another implementation of the bag of tasks which does not use STM at all. For the benchmarks of the STM implementation, we always used the variant with a common buffer for all worker threads, since this resembles the other implementation best. This variant has been shown to be slower compared to the variant with multiple task buffers.

In some benchmarks like our *Last* benchmark, the implementation without STM outperforms the other implementation (see figure 6.20). However, these are typically benchmarks in which parallelisation never resulted in better performance.

For other examples like *NDNums* (figure 6.21) and *PermSort* (figure 6.22) the simple implementation results in worse performance compared to the STM implementation. As this is the case for most of the benchmarks, we give up the idea of an implementation without STM for now. Our STM implementation seems to be efficient enough for our purposes.

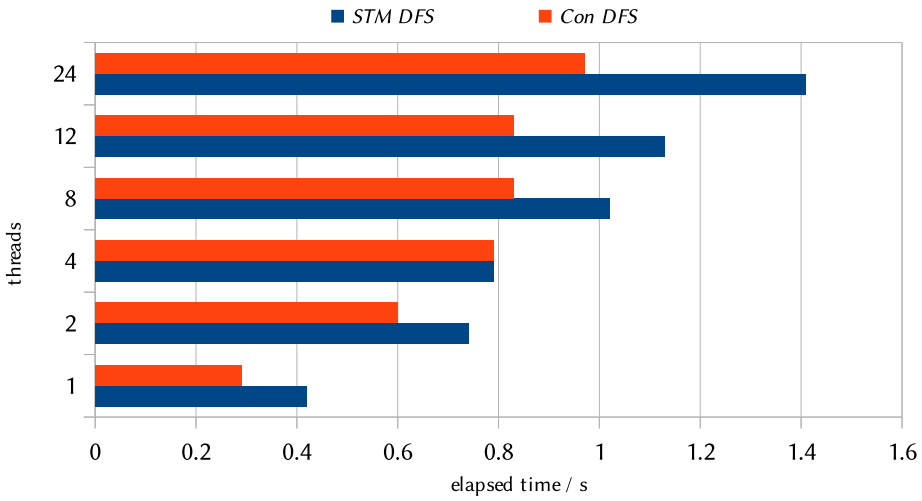


Figure 6.20.: Elapsed time on computing *one* solution of the *Last* benchmark. Comparing depth-first bag search using STM (STM DFS) with depth-first bag search without STM (Con DFS).

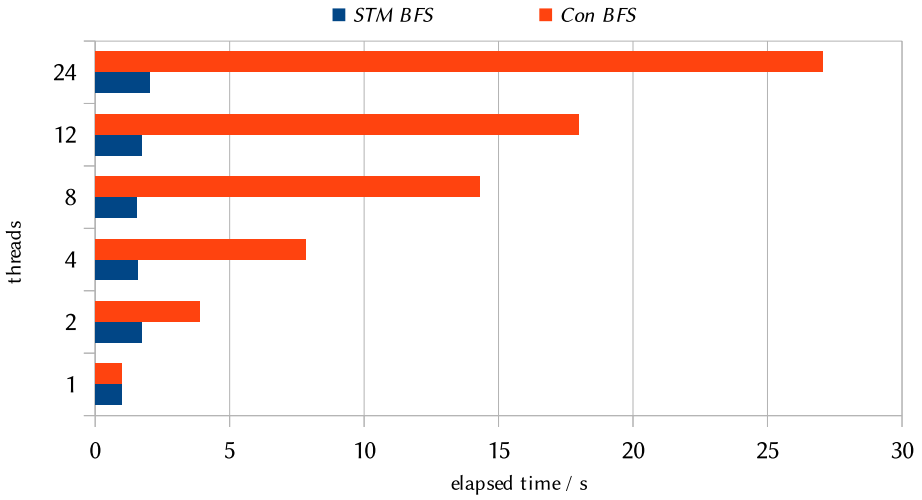


Figure 6.21.: Elapsed time on computing *one* solution of the *NDNums* benchmark. Comparing breadth-first bag search using STM (STM BFS) with breadth-first bag search without STM (Con BFS).

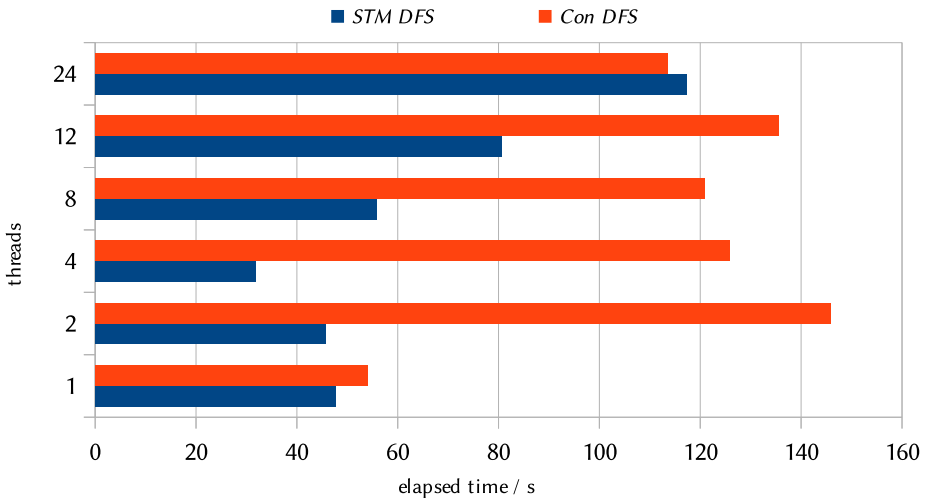


Figure 6.22.: Elapsed time on computing *all* solutions of the *PermSort* benchmark. Comparing depth-first bag search using STM (STM DFS) with depth-first bag search without STM (Con DFS).

Profitability of the Reduction of Created Sparks and Tasks

Even though a large search space with many independent computations seems to support parallelisation, each spark and each task comes with a certain overhead. Having less but larger independent computations results in less overhead, but it may also result in having not enough tasks for all threads. It is important to strike the right balance between fine and coarse granulation. This section is about estimating the benefit from reducing the number of created sparks and tasks respectively as proposed in section 4.3.3 for using sparks and in section 4.4.2 for the bag of tasks approach.

To benchmark how much speedup can be expected with this reduction, we use a variant of the *permutation sort* benchmark. Our variant uses a different definition of *permute* which is semantically identical to the implementation above, but results in a different search tree. This search tree is better balanced in comparison to the other definition. The idea making sure to get a mostly balanced tree is to make sure that each choice contains subtrees of a similar size. Here, each choice decides whether the first element of the permuted list is from the first or from the second half of the list. The other half is being rejected. Selecting an element of the remaining elements is being done with the same principle. This is slower compared to the normal permutation, but results in a better balanced list. Figure 6.23 shows the code of the permutation function.

Furthermore, we benchmark the sorting of the list $[1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2]$, supporting the

```
permute xs =
  permute' xs []
where
  permute' xs ys =
    case (xs, ys) of
      ([], []) → []
      ([], _)  → permute' ys []
      ([x], _) → x : permute' ys []
      _        →
        let xl = length xs
            (xs1, xs2) = splitAt (xl `div` 2) xs
        in permute' xs1 (xs2 ++ ys) ? permute' xs2 (xs1 ++ ys)
```

Figure 6.23.: The Code of *permute* creating a more balanced list.

balance of the tree while not making the sorting trivial as for [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]. The resulting search tree is very well-balanced as shown in figure 6.24 for level three.

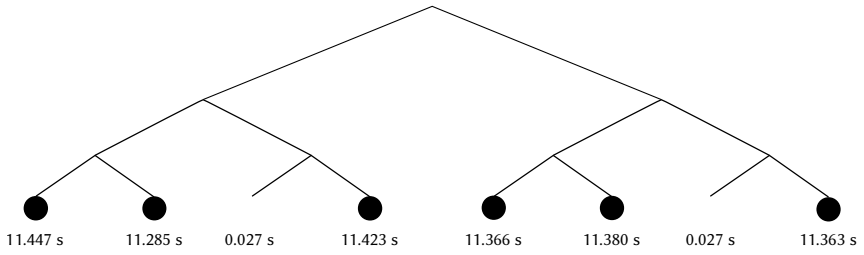


Figure 6.24.: Measuring the evaluation times of the subtrees at level three for the specially modified permutation sort. The radius of the circles increases linearly with the evaluation time of each subtree.

This program is constructed to provide a good search tree for the use with *splitLimitDepth* and *dfsBagLimit*, which are used in the benchmarks here. The diagram in figure 6.25 shows the difference in evaluation time between *splitLimitDepth* (−1) without a depth limit and the limited parallelisation strategies. Only splitting up to level one and two is unprofitable using 8 threads and more, because there are only two respectively four independent sets of work available. The best runtime, 32.97*s* has been achieved using 8 threads and a splitting depth of 4 which competes against 42.72*s* for infinite splitting. This shows that, choosing the right strategy, it is possible to decrease the runtime significantly.

Figure 6.26 shows a similar benchmark. We use *dfsBagLimit* instead of *splitLimitDepth* here; it shows the difference in evaluation time between *dfsBagLimit* (−1) without a depth limit and *dfsBagLimit* with a depth limit. Therefore, we benchmark the overhead of creating tasks. Having a look at the value for 2 threads shows that limiting the depth of parallelisation to 4

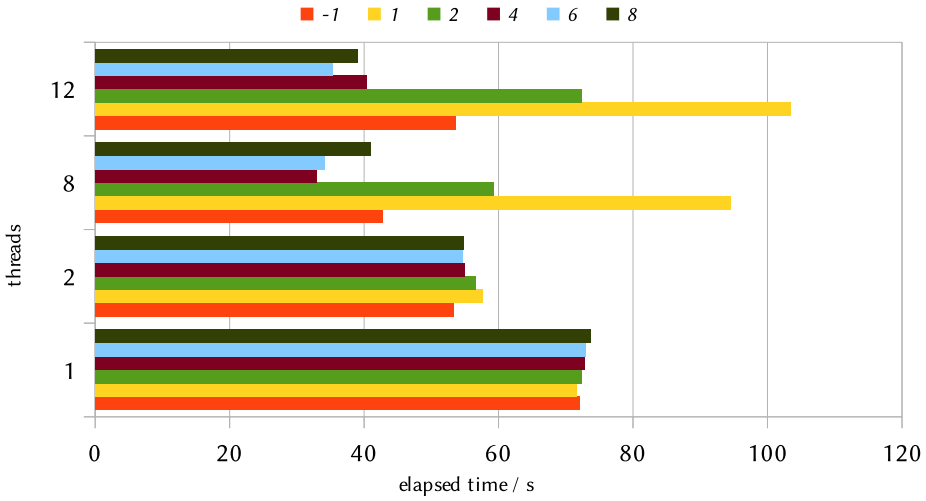


Figure 6.25.: Reducing the depth of parallelisation with *splitLimitDepth* evaluating the balanced variant of permutation sort. The bar numbers are the arguments to *splitLimitDepth* specifying the depth of parallelisation.

decreases the elapsed time from 59.31*s* to 54.01*s*. Thus, it is also profitable when using the bag of tasks approach.

As the benchmarks for both search strategies show, reducing the number of created sparks or tasks may sometimes be profitable if all parameters have been adjusted correctly. Further benchmarks have been made to compare the different strategies for reducing the number of sparks or tasks on programs, that have not been especially designed to have a balanced search tree. Even though these do not show encouraging results, it still may be possible to find other strategies with successful results.

Reducing the Number of Created Sparks and Tasks

We offer three techniques to reduce the number of created sparks and tasks. *splitLimitDepth* and *dfsBagLimit* reduce the depth to which the search strategy introduces parallelisation. This technique is not successful in any of our benchmarks. Though, *splitAlternating* as well as *splitRight* and *dfsRight* have been successful in rare cases. In the following, we have a closer look at the benchmarks of *PermSort* and *SearchQueens* evaluated with *splitRight* and *dfsRight*.

Figure 6.27 shows the evaluation of *SearchQueens* with the deterministic parallel depth-first search variant *splitRight*. Especially when splitting normally up to level 8 (dark green), we

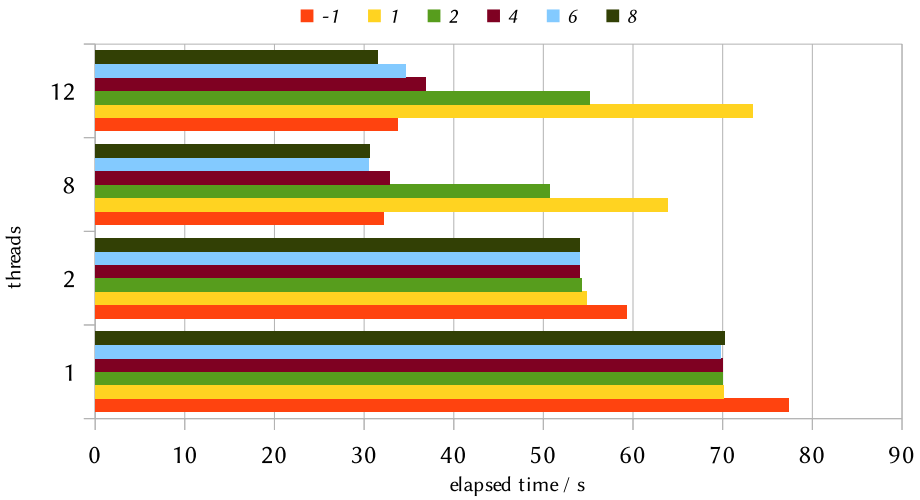


Figure 6.26.: Reducing the depth of parallelisation with *dfsBagLimit* evaluating the balanced permutation sort. The names of the series are the arguments given to *dfsBagLimit*.

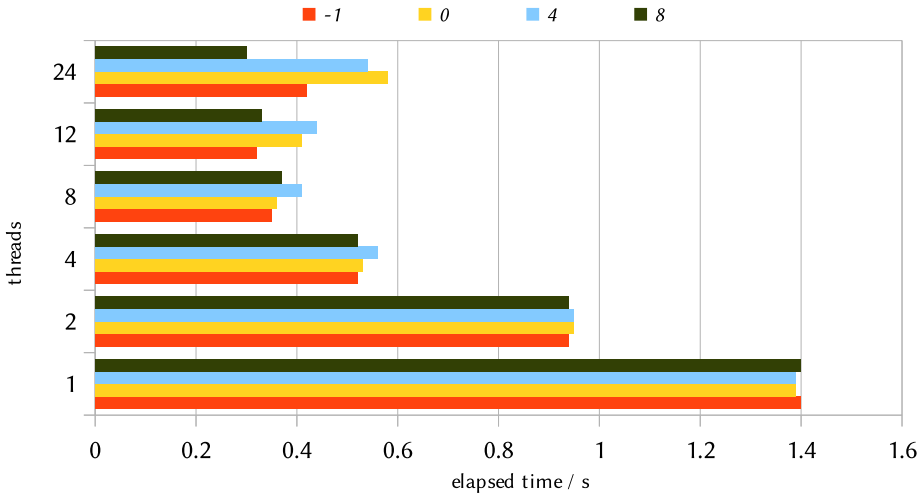


Figure 6.27.: Reducing the depth of parallelisation with *splitRight* evaluating *all* results of *SearchQueens*. The names of the series are the arguments given to *splitRight*.

can observe a performance improvement on 24 threads from 0.42*s* to 0.3*s*. For all other thread numbers, *splitRight* is only on par with the unlimited splitting strategy.

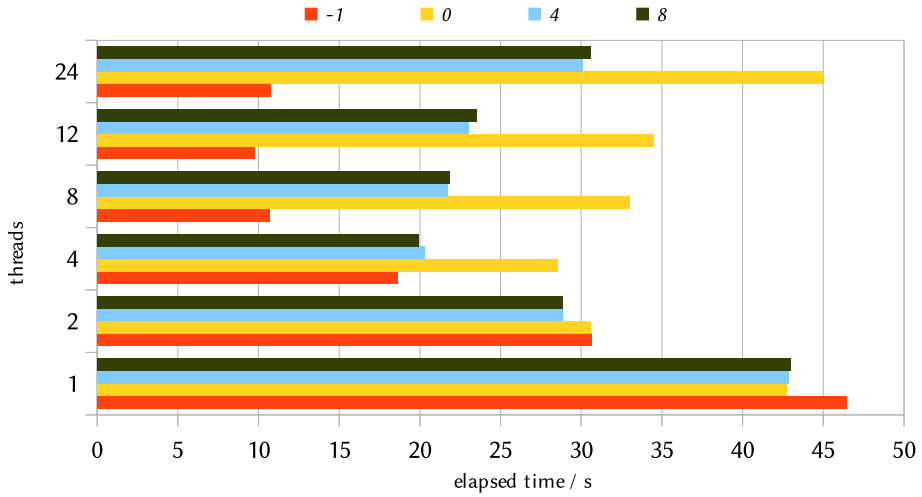


Figure 6.28.: Reducing the depth of parallelisation with *splitRight* evaluating *all* results of *PermSort*. The names of the series are the arguments given to *splitRight*.

Figure 6.28 shows the results of the evaluation of all *PermSort* results. Here, *splitRight* performs worse with a low limit number. However, we observe a minor runtime improvement for 2 threads.

The results for *dfsBagRight*, the bag of tasks variant of *splitRight*, look similar. Both the benchmark results of *SearchQueens* (figure 6.29) and *PermSort* (figure 6.30) only show a very little reduction improvement for up to 4 threads.

For our benchmark programs, none of the spark and task reduction techniques could be proven to be useful. To improve the runtime significantly, we probably need more information about the search tree of the given expression. At this time, we can not recommend any of these strategies.

Influence of Stack Size on the Memory Consumption of the Fair Search Strategies

First benchmarks of the fair search strategies showed, that all of these used a very large amount of memory. This often resulted in *out of memory* errors. After some analysis of the size of the search tree it was clear, that the size of the search tree was not the reason for this. Rather, the large demand of memory resulted from the memory each search thread allocated. With *fairSearch*1, permutation sorting a list of 12 elements results in around 270,000

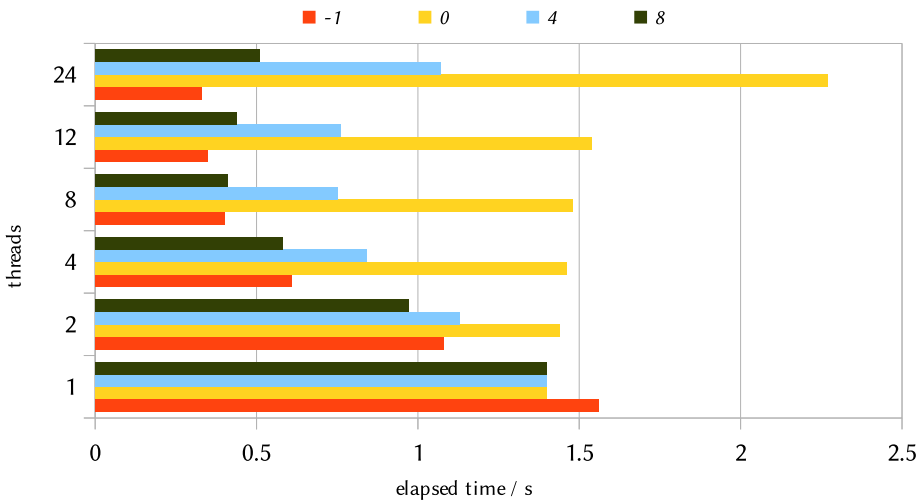


Figure 6.29.: Reducing the depth of parallelisation with *dfsBagRight* evaluating *all* results of *SearchQueens*. The names of the series are the arguments given to *dfsBagRight*.

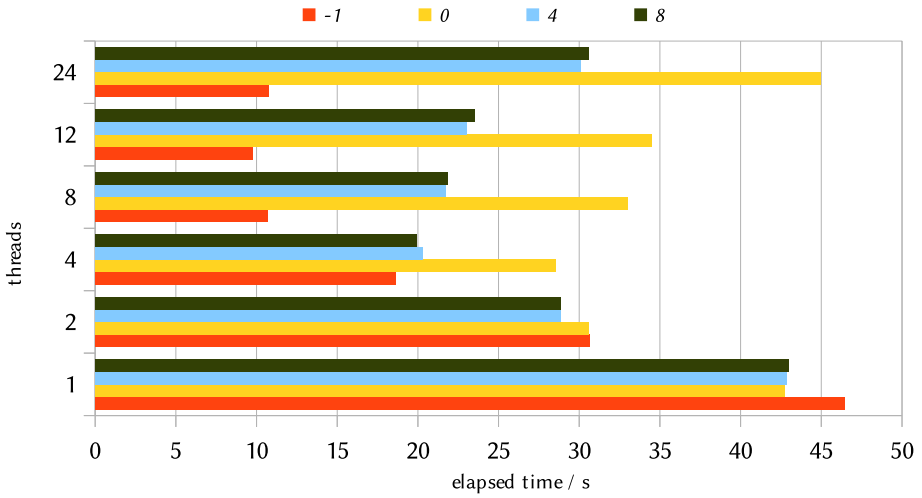


Figure 6.30.: Reducing the depth of parallelisation with *dfsBagRight* evaluating *all* results of *PermSort*. The names of the series are the arguments given to *dfsBagRight*.

threads existing at the same time. The memory needed by one thread consists of its stack size and a few bytes of information for the runtime system. The default starting stack size of GHC's runtime system is 1024 bytes, which would also not justify the huge memory consumption. This is a result of the default *stack chunk* size of 32,768 bytes. A stack chunk is the new part of the stack allocated once an existing stack overflows. During the evaluation of permutation sort, nearly all stacks overflowed, which results in more than $8.2GiB$ of memory being allocated.

As a result, we assume that increasing the initial stack size would lower the total memory consumption. The following command sets the initial stack size of threads in GHC's runtime to $2kiB$.

```
:set rts -ki2048
```

Stack	SearchQueens		PermSort	Half		Last	
Size	One	All	One	One	All	One	All
1.00	204	362	1619	52	80	109	107
1.25	206	360	**714**	51	80	117	109
1.50	**114**	**116**	811	**18**	**23**	**61**	**51**
1.75	121	124	838	19	25	65	60
2.00	126	132	938	22	29	115	126
3.00	158	167	1451	24	31	139	118
4.00	118	134	994	23	30	86	85

Table 6.2.: Benchmarking the maximum memory consumption in MiB for the search strategy *fairSearch*1 as a function of the initial stack size in kiB. 12 system threads.

To confirm this assumption and to find a better default initial stack size, we benchmark the total memory consumption as a function of the initial stack size for each thread. Table 6.2 shows the results of these benchmarks for the search strategy *fairSearch*1. For the benchmark *Half All* the memory consumption drops from $80MiB$ for an initial stack size of $1kiB$ to $23MiB$ for an initial stack size of $1.5kiB$. All in all, $1.5kiB$ can be assumed to be a good default stack size for *fairSearch*1.

Table 6.3 executes the same benchmarks with *fairSearch*2 instead of *fairSearch*1. Here, the results are less clear. Often using $1.5kiB$ of initial stack space is the best option, too. Just in *SearchQueens One* using $1.75kiB$ is significantly better. We also choose $1.5kiB$ for further benchmarks with this strategy.

For *fairSearch*3 the initial stack size does not seem to have a large influence on the maximum amount of memory used as shown in table 6.4. Just *PermSort* drops from $797MiB$ to $244MiB$ when increasing the initial stack size to $1.25kiB$. Therefore, we choose $1.25kiB$ as the initial stack size here.

Stack	SearchQueens		PermSort	Half		Last	
Size	One	All	One	One	All	One	All
1.00	FAILED	163	825	87	85	155	155
1.25	100	182	**166**	91	90	170	171
1.50	62	89	186	**33**	**31**	**128**	**122**
1.75	**36**	**88**	199	37	34	140	132
2.00	54	87	213	39	38	167	167
3.00	43	106	370	53	49	233	184
4.00	63	105	303	47	44	143	143

Table 6.3.: Benchmarking the maximum memory consumption in MiB for the search strategy *fairSearch*2 as a function of the initial stack size in kiB. 12 system threads.

Stack	SearchQueens		PermSort	Half		Last	
Size	One	All	One	One	All	One	All
1.00	**31**	152	797	52	85	**130**	127
1.25	32	**138**	**244**	52	85	139	135
1.50	33	146	323	53	**78**	134	**132**
1.75	36	162	267	53	82	143	140
2.00	43	169	354	**51**	80	169	168
3.00	54	175	366	59	90	180	174
4.00	40	161	337	54	79	150	148

Table 6.4.: Benchmarking the maximum memory consumption in MiB for the search strategy *fairSearch*3 as a function of the initial stack size in kiB. 12 system threads.

Comparing Fair Search Strategies

After having chosen a default stack size for each fair search strategy, we compare these in terms of memory consumption and runtime. The sequential reference is breadth-first search, which does not feature the same level of completeness as the fair search strategies. Therefore, being faster than this sequential breadth-first search is not the primary goal here.

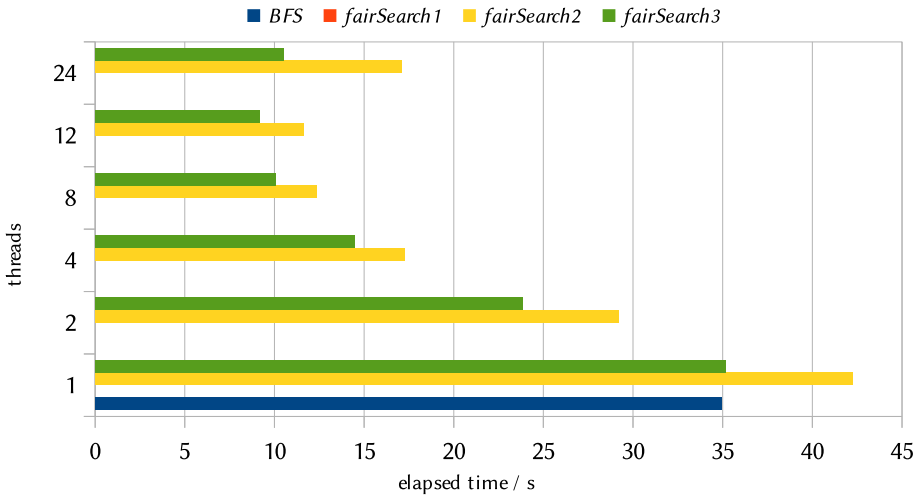


Figure 6.31.: Diagram of the elapsed time on running the *EditSeq* benchmark. Comparing the fair search strategies. *fairSearch*1 fails at this benchmark.

First of all, for some programs like *EditSeq* and *PermSort*, *fairSearch*1 does not complete the search in 1000 seconds due to an extensive memory usage. One example can be seen in figure 6.31. Therefore, we consider *fairSearch*1 to be not usable in praxis.

Having a look at the other strategies *fairSearch*2 and *fairSearch*3, both result in better performance than breadth-first search. In *EditSeq* (figure 6.31), *fairSearch*3 performs slightly better than *fairSearch*2 with 9.16*s* in comparison to 11.63*s* at 12 threads. However, the memory consumption of *fairSearch*3 is clearly higher than the memory consumption of *fairSearch*2 as shown in figure 6.32.

For the benchmark *Half* (figures 6.33 and 6.34), *fairSearch*1 computes results and it is also not significantly slower for all thread counts. Also the memory usage of *fairSearch*1 is comparable to the memory usage of *fairSearch*2 while the usage of *fairSearch*3 is significantly higher.

Figure 6.35 shows the memory usage while computing all values of the *SearchQueens* benchmark. Here, both *fairSearch*1 and *fairSearch*3 show a huge memory usage whereas *fairSearch*2

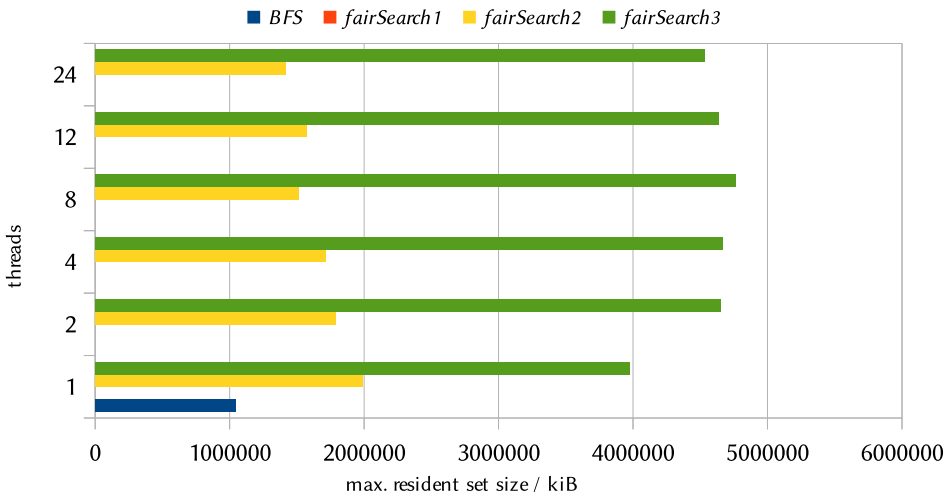


Figure 6.32.: Diagram of the maximum memory consumption on running the *EditSeq* benchmark. Comparing the fair search strategies. *fairSearch*1 fails at this benchmark.

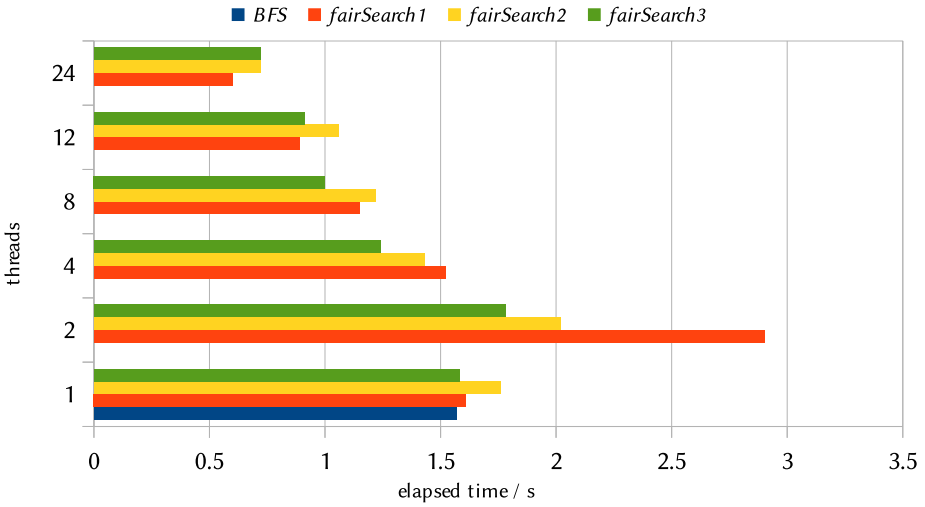


Figure 6.33.: Diagram of the elapsed time on computing *all* values of the *Half* benchmark. Comparing the fair search strategies.

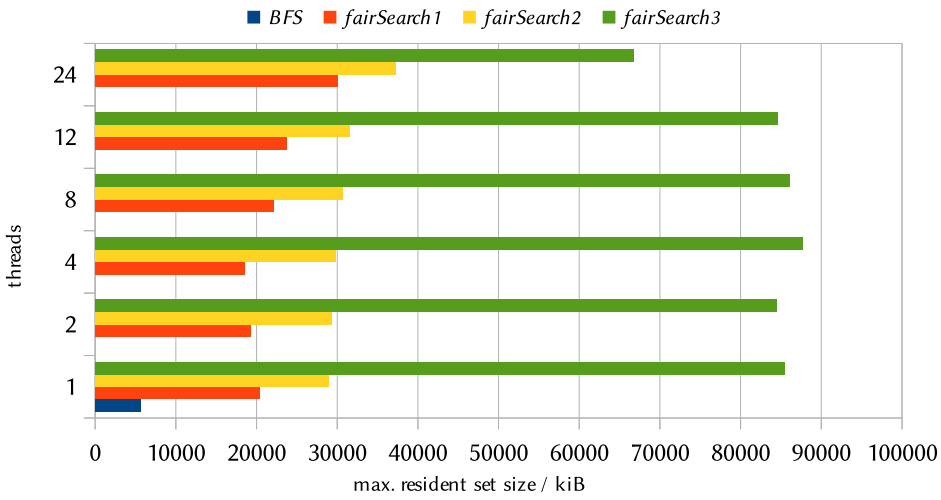


Figure 6.34.: Diagram of the maximum memory consumption on computing *all* values of the *Half* benchmark. Comparing the fair search strategies.

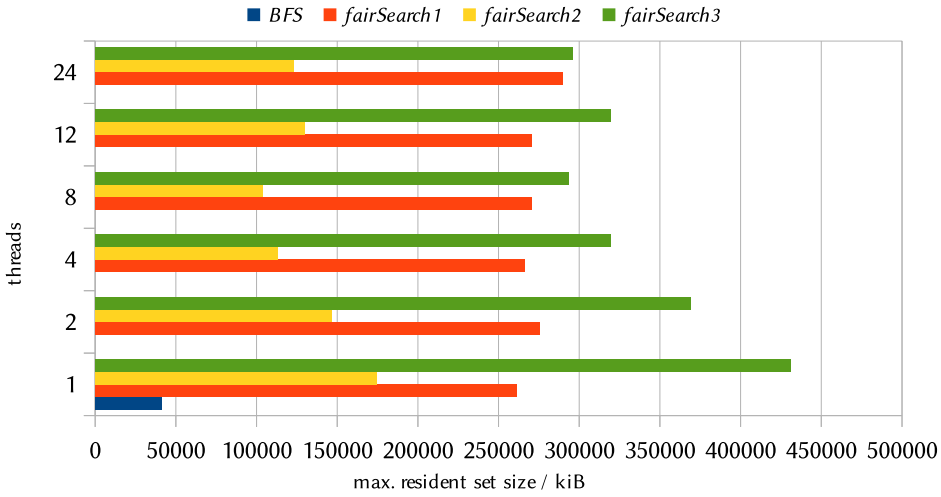


Figure 6.35.: Diagram of the maximum memory consumption on computing *all* values of the *SearchQueens* benchmark. Comparing the fair search strategies.

is still near to the memory usage of sequential breadth-first search.

All in all, *fairSearch*2 seems to be the most reasonable fair search strategy. It finishes all benchmarks with a decent performance and a low memory usage.

6.4. Summary

This section summarises the results of the evaluation. Table 6.5 gives a general overview of the performance of all implemented parallel search strategies. Because of the different completeness behaviour, the table is divided into three parts: strategies completing with depth-first search, strategies competing with breadth-first search, and fair search strategies.

The columns for *runtime* and *memory* show the runtime behaviour of the search strategies. Here, one or multiple "+" symbols indicate, that this strategy achieved an improvement over the sequential search, the symbol "−" indicates a decline, and "○" indicates no signifant change. Furthermore, "++" indicates that the strategy can be used reasonably and strategies marked with a "+++" are among the best of their category. These can be thought of as a default parallel search strategy. The column *abortion* shows whether the abortion of the search works correctly as verified in section 6.2. If the table cell shows the symbol "✓" in the line of a search strategy, the computation of the values has been aborted successfully.

Complete benchmark results of the most successful strategies can be found in appendix B. These compare the sequential strategies with the parallel strategies *splitAll*1, *dfsBag takeFirst*, *fdfsBag takeFirst*, *bfsParallel*2, *bfsBag takeFirst*, and *fairSearch*2.

Name	Runtime		Memory	Abortion
	One	All		
Strategies Competing with Depth-First Search				
splitAll1/parSearch	−	+++	○	
splitAll2	−	+	○	
splitAll3	−	++	○	
splitLimitDepth	−	○	○	
splitAlternating	○	○	○	
splitRight	−	++	○	
splitLeft	−	○	○	
dfsBag (common buffer)	+	+	○	✓
dfsBag (take first)	+++	+++	○	✓
dfsBag (split half)	+++	+++	○	✓
dfsBag (split vertical)	+++	+++	○	✓
dfsBag w/o STM (common buffer)	−	−	○	✓
fdfsBag (common buffer)	+	+	−	✓
fdfsBag (take first)	+++	+++	−	✓
fdfsBag (split half)	○	○	−	✓
fdfsBag (split vertical)	○	○	−	✓
fdfsBag w/o STM (common buffer)	−	−	−	✓
dfsBagLimit (take first)	○	○	○	✓
dfsBagRight (take first)	○	++	○	✓
dfsBagLeft (take first)	○	○	○	✓
Strategies Competing with Breadth-First Search				
bfsParallel1	○	−	○	
bfsParallel2	○	++	○	
bfsParallel3	○	++	−	
bfsBag (common buffer)	+	+	○	✓
bfsBag (take first)	+++	+++	○	✓
bfsBag (split half)	+	+	○	✓
bfsBag (split vertical)	+	+	○	✓
bfsBag w/o STM (common buffer)	+	+	−	✓
Fair Search Strategies				
fairSearch1	−	−	−−	✓
fairSearch2	++	++	−	✓
fairSearch3	++	++	−−	✓

Table 6.5.: Comparison matrix including all implemented parallel search strategies.

7. Usage of Parallel Search

The search strategies presented in this thesis are accessible in multiple ways. All parallel strategies have been made available through the Curry library *ParallelSearch*, which allows the programmer to access parallel search out of the Curry program in *IO* code. It is also possible to use these strategies as the top-level search strategy which is used for the main program. Another way to use some of these parallel search strategies is via the set functions (see section 2.1.6), which allows using parallel search in pure code.

After starting KiCS2, it shows a command prompt which allows executing programs and to set various configuration options. It offers a basic overview over these configuration options after entering the command `:set`.

To exploit multiple processor cores, it is necessary to run the

```
:set threads [n]
```

command where `[n]` has to be substituted by the number of system threads to be used by the runtime system. Often it is a good idea to set the number of threads to the number of processor cores in the system. See the results of the benchmarks in section 6.3.3 for more information. Omitting the number of threads lets the runtime choose n itself depending on how many processor cores are in your machine.

After setting the desired configuration options, you might want to execute your program by entering the main goal on the command prompt. To measure the speedup, it is necessary to see the execution time.

```
:set +time
```

Top-Level Search

The top-level search strategy can be set to depth-first search (`dfs`), breadth-first search (`bfs`), or fair search (`fair`) with the following commands:

```
:set dfs
:set bfs
:set fair
```

The fair search strategy is parallel by default; otherwise, you need to turn on the parallel top-level search with the `parallel` flag.

```
:set +parallel
```

Using the flag `fixed`, it is possible to select between search strategies with a fixed value order and those with an order which depends on the evaluation order.

```
:set -fixed
```

The above command switches off enforcing the fixed order, which is sometimes faster, when requesting only one value. Switching off the fixed order is equivalent to using a bag of tasks strategy.

By default, KiCS2 prints out all solutions for the main goal, but it is possible to compute only one solution or browse though the solutions interactively.

```
:set +first
:set +interactive
```

Remember to activate one of these options and a complete search strategy like breadth-first search or fair search if you want to run a search for solutions in an infinite search tree.

ParallelSearch Library

Often, you might want to use non-deterministic computations as part of a deterministic main program. For parallel search strategies, this is possible with the Curry library *ParallelSearch*. The interface mostly consists of the following functions.

$$
\begin{aligned}
&\mathit{getOneValue} &&:: \mathit{Strategy} \to a \to \mathit{IO}\ (\mathit{Maybe}\ a)\\
&\mathit{getLazyValues} &&:: \mathit{Strategy} \to a \to \mathit{IO}\ [a]\\
&\mathit{getAllValues} &&:: \mathit{Strategy} \to a \to \mathit{IO}\ [a]
\end{aligned}
$$

All of these functions are parameterised with a search strategy. Executing these actions starts the parallel search with the given strategy. While *getOneValue* only computes the first result, similar to setting the option `+first` of the top-level search, *getAllValues* and *getLazyValues* compute all results. The difference between *getAllValues* and *getLazyValues* is that *getAllValues* evaluates the complete list before returning it whereas *getLazyValues* defers the evaluation of the list lazily. In most cases *getLazyValues* is the function of choice as it allows processing the results in parallel to their computation. For example when we want to sort the list of results.

$$
\begin{aligned}
&\mathit{main} =\\
&\quad \mathit{vlist} \leftarrow \mathit{getLazyValues}\ \mathit{splitAll}1\ \mathit{goal}\\
&\quad \textbf{let}\ \mathit{sorted} = \mathit{sortBy}\ (\leq)\ \mathit{vlist}\\
&\quad \mathit{return}\ \mathit{sorted}
\end{aligned}
$$

Set Functions

Set functions allow using encapsulated search in pure code as shown in section 2.1.6. Normally, search strategies for set functions are defined in Curry itself, but at the moment it is not possible to write parallel search strategies in Curry. However, since KiCS2 allows defining Curry functions in Haskell, it is possible to define some of the parallel search strategies for set functions as well. Among these is a deterministic parallel search strategy *parDfsStrategy* similar to *splitAll*1, *parBfsBagStrategy* similar to *bfsBag*, and *parDfsBagStrategy* which is similar to *dfsBag*.

8. Future Work

While this thesis already presented viable parallel search strategies, there is still room for improvement in terms of runtime. The benchmarks show that both tasks and sparks have a certain overhead. Reducing the number of created tasks and sparks is not profitable for realistic example programs. Possible ways to solve this problem are presented in section 8.1. Certain Prolog implementations use *AND-parallelism* to decrease evaluation times, which is presented in section 8.2. Iterative deepening search is another sequential search strategy that can be implemented in a parallel manner. A first approach of parallel iterative deepening search is presented in 8.3.

8.1. Manual and Automatic Annotation

While we were successful decreasing the computation time of a specifically designed example program, using these strategies on real programs did not result in better runtimes. These strategies limited the creation of threads and tasks by limiting the tree depth of parallelisation, only using parallelisation in certain hand sides of the search tree, and arbitrarily parallelising the evaluation of every second subtree. All these strategies tried to reduce the number of created sparks or tasks without taking the code of the program into account. Ideally, small parts of the evaluation that do not take much time should be evaluated in sequence while multiple larger parts could be evaluated in parallel.

Often, while writing a program, the developer already knows that certain parts of the program would not take much time in the evaluation process. The programmer could then annotate those expressions of which he thinks the evaluation in parallel would not be reasonable. Using *SetFunctions*, this is already possible.

$$chooseValue :: Values\ a \to a$$

chooseValue chooses an arbitrary value of the given set non-deterministically. In combination with one of the set functions like *set*1 and *set*1*With* it is possible to annotate that the non-determinism introduced by a given function should not be evaluated in parallel.

$$withDfs1 :: (a \to b) \to a \to b$$
$$withDfs1\ f\ x = chooseValue\ (set1\ f\ x)$$

Semantically, *withDfs*1 is the identity on functions with one parameter as it encapsulates the non-determinism and then returns all these results non-deterministically, but due to the use

of *set*1, the evaluation of these values is done in sequence with depth-first search. However, the parallelism introduced by *chooseValue* may then result in a parallel evaluation. Dead ends with failures are thereby removed.

In contrast to specifying that a certain part of the program should not be evaluated in parallel, it is also possible do the opposite: specifying that a certain evaluation should be done in parallel.

$$withParDfs1 :: (a \to b) \to a \to b$$
$$withParDfs1\ f\ x = chooseValue\ (set1 With\ parDfsStrategy\ f\ x)$$

Note that using *chooseValue* on the result of a set function forces the evaluation of the values to normal form. While this annotation is already possible with the current implementation, we did not do any benchmarking on it. This will have to be done by future users of parallel evaluation in Curry.

Annotating the whole program in this way can be time-consuming and annoying. Possibly, this annotation can be done automatically either dynamically during the runtime or by analysing the source code statically. Similar automatic annotation is done by the *Ciao* Prolog system [12].

8.2. Prolog's AND-Parallelism

For other non-deterministic programming languages like Prolog, two types of parallelism have been identified and successfully exploited: *AND-parallelism* and *OR-parallelism*. In this thesis we implemented OR-parallelism. We describe AND-parallelism with the help of Prolog programs. Prolog programs consist of clauses of the following form:

$$h :- g_1, g_2, ..., g_n.$$

The above clause means: if all the terms $g_1, g_2, ..., g_n$ are true, then h is also true, where the term on the left hand side of :–, h, is called *head*. These terms can either be a simple term like the atom *husain* or a compound term like *father*(*gaurav*, *husain*) or *parent*(X, Y), where identifiers starting with an uppercase letter are variables. A clause for the *parent* relation might be the following:

$$parent(X, Y) :- father(X, Y).$$
$$parent(X, Y) :- mother(X, Y).$$

The above clauses are also an example for one head which has multiple clauses, which can be read as the disjunction of both clauses "X is a parent of Y if it is the father of Y or X is a parent of Y if it is the mother of Y". In contrast, the comma between multiple goals of the same clause is read as the conjunction between these. The following clause can be read as "X and Y have children together if there is a Z for which X is the mother of Z and Y is the father of Z":

$$have\_children\_together(X, Y) :- mother(Z, X), father(Z, Y).$$

Evaluating multiple clauses for one head in parallel and therefore parallelising the evaluation of different alternatives is called OR-parallelism. This is equivalent to evaluating two or more Curry expressions combined with choice operator in parallel.

AND-parallelism, the second type of parallelism exploited by Prolog systems, arises when a clause contains more than one goal and the resolution of these subgoals is done in parallel.

There are two distinct classes of AND-parallelism. *Independent AND-parallelism* only resolves subgoals in parallel if they do not share any free variables and are therefore independent. *Dependent AND-parallelism* resolves all subgoals in parallel, even if two or more of these subgoals share free variables. The implementation of dependent AND-parallelism is more complicated, since the binding of a variable in one branch also affects the binding of the variable in the other branch.

Unfortunately, Curry's programs differ from those written in Prolog and are more similar to programs written in functional programming languages. However, it is also possible to write programs similar to those written in Prolog. The following rule is a direct translation of the rule *have_children_together* defined in Curry:

```
have_children_together x y | mother z =:= x & father z =:= y = success
  where z free
```

This rule may be parallelised similar to AND-parallelism in Prolog by evaluating the both rules combined with the & operator in parallel.

Another definition containing a conjunction in Prolog is that of the grandfather relation.

```
grandfather(X, Y) :- father(X, Z), father(Z, Y).
grandfather(X, Y) :- mother(X, Z), father(Z, Y).
```

The equivalent Curry code might look like the following code, which may be parallelised similarly to the Prolog code:

```
grandfather x | father   x =:= z & father z =:= y = y
  where y, z free
grandfather x | mother x =:= z & father z =:= y = y
  where y, z free
```

However, most Curry programmers would write this program in the following way. Note that this program is equivalent to the above and would look more common to functional programmers:

```
grandfather x = father (father   x)
grandfather x = father (mother x)
```

At this point, the approach to evaluate the operands of & in parallel would not be profitable anymore and the evaluation would fall back to a sequential evaluation.

However, it is also possible to understand the functional definition of *grandfather* in a way similar to the Prolog-style variant. We could decide to evaluate the argument and its application in terms of dependent AND-parallelism. Then, the major difference between Prolog and Curry is the aspect of laziness. We never know much of the argument has to be evaluated. Evaluating the argument to normal form in parallel would often be unnecessary if the value of the argument is only partly demanded. As a result, we would depend on annotations of the programmer or of an automatic code-analysis. Further reasoning about AND-parallelism and its implementation is left for future publications.

8.3. Parallel Iterative Deepening Search

In this thesis, we implemented parallel search strategies similar to depth-first search, those similar to breadth-first search, and fair search strategies. In contrast to this, we did not take iterative deepening search into account. Iterative deepening is complete for search trees without infinite deterministic computation similar to breadth-first search. This is reached by searching the tree with depth-first search with an additional depth limit. When the whole search tree up to the depth limit has been searched but the tree is higher than the given limit, the search is restarted with a higher depth limit. A sketch of the implementation of iterative deepening can be seen in figure 8.1.

```
idsSearch :: Int → (Int → Int) → SearchTree a → [a]
idsSearch initdepth incr st = iterIDS initdepth (collectBounded 0 initdepth st)
  where
  iterIDS _ Nil          = []
  iterIDS n (Cons x xs) = x : iterIDS n xs
  iterIDS n Abort        = let newdepth = incr n
                           in  iterIDS newdepth (collectBounded n newdepth st)
collectBounded :: Int → Int → SearchTree a → AbortList a
collectBounded oldbound newbound st = collectLevel newbound st
  where
  collectLevel _ None = Nil
  collectLevel d (One x)
    | d ≤ newbound − oldbound = x `Cons` Nil
    | otherwise               = Nil
  collectLevel d (Choice x y)
    | d > 0      = collectLevel (d − 1) x +! collectLevel (d − 1) y
    | otherwise = Abort
```

Figure 8.1.: Sketching the implementation of iterative deepening search.

Normally, iterative deepening offers a better space complexity compared to breadth-first search as it does not have to save a whole level of the search tree at once. However, for

the code presented in figure 8.1, the Haskell runtime system keeps the whole search tree in memory. How to reach the desired space complexity is omitted here.

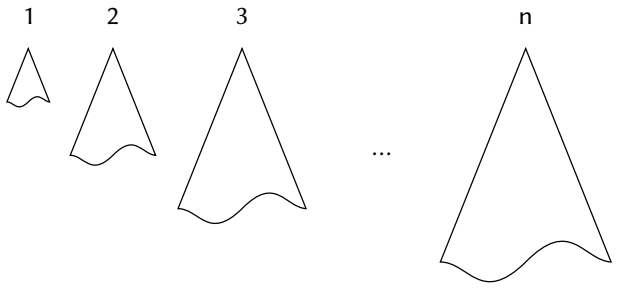


Figure 8.2.: A schematic sketch of parallel iterative deepening search.

The parallel iterative deepening strategy proposed here is similar to that used by Powley and Korf [29]. Here, each thread does a depth-first search up to a certain depth. At the same time, the other threads do depth-first search up to higher depth limits. This is shown schematically in figure 8.2.

9. Conclusion

As part of this thesis, we integrated various parallel search strategies into the Curry system KiCS2. These allow evaluating a non-deterministic program or expression in parallel without any kind of change or annotation of the source code. While it will still be possible to improve the strategies, we already reached significant speedups for both depth-first and breadth-first search strategies. Furthermore, we presented a fair search strategy, whose purpose, instead of improving the runtime, is to provide a complete search for programs with deterministic loops.

Basically we implemented three kinds of search strategies. The deterministic parallel search strategies preserve the order of values in comparison to their sequential counterparts. The strategies using the bag of tasks approach have a similar behaviour in terms of completeness, but return their values in an arbitrary order depending on the scheduling. Fair search strategies form the third group.

All these strategies were compared with each other in terms of memory usage and runtime. Performing benchmarks and analysing their results was very time-consuming; especially, because little change to the code often has large impact on the performance.

The results of these benchmarks are encouraging. The deterministic parallel depth-first search strategy *splitAll*1 usually reaches speedups of factor four with eight threads on computing all values of an expression. However, it is not profitable on computing only one value of an expression which yields multiple results. The deterministic breadth-first search strategy *bfsParallel*2 reaches a speedup of factor two with eight threads. The bag of tasks strategies have been more successful on breadth-first search: *bfsBag takeFirst* often reaches a speedup of factor four with twelve threads. Additionally, using *dfsBag takeFirst* or *fdfsBag takeFirst* results in a similar speedup of factor four compared to depth-first search. Also, all bag of tasks strategies are profitable when computing only one value of an expression.

These speedups are smaller than one might expect from the theory. Especially, the performance rarely profits from using more than twelve threads. To improve the strategies further, we tried using alternative work-splitting approaches. While thereby reducing the number of parallelisation chunks has been proven to be useful for a specific good-case example. Unfortunately, it was not possible to reach reliable speedups for normal programs. This has been identified to be a starting-point for future work.

Due to problems with the amount of memory allocated by fair search strategies, we thought that it would not be possible to use these profitably for a long time. Fortunately, we identified, as the reason for this, the amount of memory allocated for the stack of each thread. As a result,

we now have the fair search strategy, *fairSearch*2, which is not only usable, but sometimes even faster than sequential breadth-first search.

In the end, the user is able to select from few parallel search strategies that will most likely perform well for the given program. Depending on the desired degree of completeness, this is often *dfsBag takeFirst* or *bfsBag takeFirst*. Sometimes it may be necessary to get the same order of values in every run of the program. In this case, the strategies *splitAll*1 and *bfsParallel*2 are recommended. However, for the bag of tasks strategies both the order of the results is not fixed and aborting the computation works more reliably. For programs with deterministic loops, which are often the result of errors, the strategy *fairSearch*2 produces good results. Furthermore, it is possible to use parallel search strategies with encapsulated search. This allows the user to select different suitable search strategies for different parts of the program.

Bibliography

[1] Elvira Albert, Michael Hanus, Frank Huch, Javier Oliver, and Germán Vidal. Operational semantics for declarative multi-paradigm languages . *Journal of Symbolic Computation*, 40(1):795–829, 2005. Reduction Strategies in Rewriting and Programming special issue.

[2] Sergio Antoy and Michael Hanus. Set functions for functional logic programming. In António Porto and Francisco Javier López-Fraguas, editors, *PPDP*, page 73–82. ACM, 2009.

[3] Till Berger and David Sabel. Parallelizing DPLL in Haskell. In Stefan Wagner and Horst Lichter, editors, *Software Engineering (Workshops)*, volume 215 of *LNI*, page 27–42. GI, 2013.

[4] Bernd Braßel, Michael Hanus, Björn Peemöller, and Fabian Reck. KiCS2: A New Compiler from Curry to Haskell. In Herbert Kuchen, editor, *WFLP*, volume 6816 of *Lecture Notes in Computer Science*, page 1–18. Springer, 2011.

[5] Matthias Böhm. Erweiterung von Curry um Typklassen. Master's thesis, October 2013.

[6] Amadeo Casas, Manuel Carro, and Manuel V. Hermenegildo. A High-Level Implementation of Non-deterministic, Unrestricted, Independent And-Parallelism. In Maria Garcia de la Banda and Enrico Pontelli, editors, *ICLP*, volume 5366 of *Lecture Notes in Computer Science*, page 651–666. Springer, 2008.

[7] A. M. Cheadle, A. J. Field, Simon Marlow, Simon L. Peyton Jones, and R. L. While. Exploring the Barrier to Entry: Incremental Generational Garbage Collection for Haskell. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, page 163–174, New York, NY, USA, 2004. ACM.

[8] Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, page 345–363, 1936.

[9] M. Hanus and R. Sadre. An Abstract Machine for Curry and its Concurrent Implementation in Java. Number 6. MIT Press, 1999.

[10] Michael Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8.3). Available at `http://www.curry-language.org`, 2012.

[11] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable Memory Transactions. 2005.

[12] Manuel V. Hermenegildo, Francisco Bueno, Manuel Carro, Pedro López-García, Edison Mera, José F. Morales, and Germán Puebla. An overview of Ciao and its design philosophy. *TPLP*, 12(1-2):219–252, 2012.

[13] Heinrich Hussmann. Nondeterministic algebraic specifications and nonconfluent term rewriting . *The Journal of Logic Programming*, 12(3):237–255, 1992.

[14] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Annual Symposium on Principles of Programming Languages*, page 295–308. ACM, 1996.

[15] Simon Peyton Jones, Simon Marlow, and Conal Elliott. Stretching the storage manager: weak pointers and stable names in Haskell, 1999.

[16] Miran Lipovaĉa. *Learn You a Haskell for Great Good.* April 2011.

[17] Simon Marlow. Parallel and Concurrent Programming in Haskell. In Viktória Zsók, Zoltán Horváth, and Rinus Plasmeijer, editors, *CEFP*, volume 7241 of *Lecture Notes in Computer Science*, page 339–401. Springer, 2011.

[18] Simon Marlow et al. Documentation of "System.Mem.Weak". `http://hackage.haskell.org/package/base-4.6.0.1/docs/System-Mem-Weak.html`. [Online; accessed 22-May-2014].

[19] Simon Marlow et al. GHC ticket "Weak pointer to MVar is finalized, even though MVar is still accessible". `https://ghc.haskell.org/trac/ghc/ticket/6130`. [Online; accessed 28-February-2014].

[20] Simon Marlow et al. Haskell 2010 language report. `http://www.haskell.org/onlinereport/haskell2010`, 2010.

[21] Simon Marlow, Tim Harris, Roshan P. James, and Simon Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap, 2008.

[22] Simon Marlow, Simon L. Peyton Jones, and Satnam Singh. Runtime support for multicore Haskell. In Graham Hutton and Andrew P. Tolmach, editors, *ICFP*, page 65–78. ACM, 2009.

[23] Simon Marlow and Simon Peyton Jones. The Glasgow Haskell Compiler. 2012.

[24] Simon Marlow, Simon Peyton Jones, Andrew Moran, and John Reppy. Asynchronous Exceptions in Haskell. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, page 274–285, New York, NY, USA, 2001. ACM.

[25] Simon Marlow, Patrick Maier, Hans-Wolfgang Loidl, Mustafa Aswad, and Philip W. Trinder. Seq no more: better strategies for parallel Haskell. In Jeremy Gibbons, editor, *Haskell*, page 91–102. ACM, 2010.

[26] Simon Marlow, Austin Seipp, Simon Peyton Jones, et al. Documentation of "Control.Exception.Base". `http://hackage.haskell.org/package/base-4.6.0.1/docs/Control-Exception-Base.html`. [Online; accessed 22-May-2014].

[27] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[28] Neill Mitchell, Simon Marlow, et al. GHC ticket "Deadlock in Chan module". `https://ghc.haskell.org/trac/ghc/ticket/4154`. [Online; accessed 6-May-2014].

[29] Curt Powley and Richard E. Korf. Single-agent parallel window search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(5):466–477, 1991.

[30] Fabian Reck and Sebastian Fischer. Towards a Parallel Search for Solutions of Non-deterministic Computations. In Stefan Fischer, Erik Maehle, and Rüdiger Reischuk, editors, *GI Jahrestagung*, volume 154 of *LNI*, page 2889–2900. GI, 2009.

[31] Philip W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon L. Peyton Jones. Algorithms + Strategy = Parallelism. *J. Funct. Program.*, 8(1):23–60, 1998.

[32] Rui Vieira, Ricardo Rocha, and Fernando M. A. Silva. On Comparing Alternative Splitting Strategies for Or-Parallel Prolog Execution on Multicores. *CoRR*, abs/1301.7690, 2013.

[33] Philip Wadler. Comprehending Monads. In *Mathematical Structures in Computer Science*, page 61–78, 1992.

[34] Philip Wadler and Stephen Blott. How to Make Ad-hoc Polymorphism Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, page 60–76, New York, NY, USA, 1989. ACM.

A. Wrong Divisor Implementation as an Application of Fair Search Strategies

The modulo operator can be implemented by subtracting the divisor from the divident as long as it is not smaller.

```
mod :: Int → Int → Int
n `mod` m | n ≥ m     = (n − m) `mod` m
          | otherwise = n
```

The operation *divisor* returns a divisor of the given number, but not 1 or the number itself.

```
divisor n | n `mod` m =:= 0 =
  if m ≡ n ∨ m ≡ 1
    then failed
    else m
  where m free
```

Note that divisor does not exclude 0 from the divisors, so 0 is also tested for being a divisor of n. This is never successful as subtracting 0 from n does not change n at all.

The following function tests if the given number is not a prime using a fair search strategy.

```
isNotPrime n = do
  one ← getOneValue fairSearch (divisor n)
  case one of
    Nothing → return False
    Just _  → return True
```

Note that by using the fair search strategy, it returns *True* for numbers that are not prime. *False* is never returned as the computation of whether 0 is a divisor of n does not terminate. Using breadth-first or depth-first search it does never return anything.

B. Benchmark Results of a Selection of Search Strategies

This appendix gives an overview over the recommended parallel search strategies and their performance on the presented benchmark programs. For each benchmark, we measure the wallclock time and the maximum amount of memory consumed. The wallclock time is measured in seconds and the memory is measured in MiB. Again, there are three categories: benchmarks that are incomplete, benchmarks that are complete for infinite search trees, and the fair strategies. In each category, the best value for a given thread count is highlighted. As *fairSearch2* is in a category of its own, nothing is highlighted here. "oom" indicates that the computation ran out of memory.

SearchQueens – One

Elapsed Time (s)

	DFS	*splitAll1*	*dfsBag*	*fdfsBag*	*BFS*	*bfsParallel2*	*bfsBag*	*fairSearch2*
1	**0.11**	**0.11**	0.16	0.33	0.44	**0.43**	0.51	2.44
2		0.15	**0.09**	0.31		0.47	**0.38**	0.68
4		0.13	**0.07**	0.13		0.36	**0.13**	0.61
8		0.16	**0.04**	0.07		0.40	**0.08**	0.32
12		0.14	**0.05**	0.07		0.48	**0.07**	0.32
24		0.16	**0.05**	0.06		0.47	**0.08**	0.41

Max. Resident Set Size (MiB)

	DFS	*splitAll1*	*dfsBag*	*fdfsBag*	*BFS*	*bfsParallel2*	*bfsBag*	*fairSearch2*
1	6	**5**	7	19	29	**28**	32	170
2		9	**7**	21		32	**25**	100
4		18	**9**	17		34	**19**	114
8		20	**10**	18		38	**19**	107
12		27	**14**	20		42	**19**	103
24		37	**19**	24		52	**26**	110

SearchQueens – All

Elapsed Time (s)

	DFS	*splitAll*1	*dfsBag*	*fdfsBag*	*BFS*	*bfsParallel*2	*bfsBag*	*fairSearch*2
1	**1.41**	**1.41**	1.53	1.85	**1.89**	**1.89**	2	2.75
2		**1.03**	1.08	1.35		1.64	**1.59**	1.94
4		**0.54**	0.60	0.86		**1.00**	1.09	1.14
8		**0.36**	0.41	0.62		**0.76**	0.80	0.88
12		**0.33**	0.35	0.53		1.28	**0.76**	0.94
24		0.50	**0.38**	0.54		**0.66**	0.76	1.06

Max. Resident Set Size (MiB)

	DFS	*splitAll*1	*dfsBag*	*fdfsBag*	*BFS*	*bfsParallel*2	*bfsBag*	*fairSearch*2
1	**10**	**10**	**10**	40	**40**	41	42	170
2		14	**12**	34		46	**35**	143
4		22	**15**	30		40	**34**	110
8		25	**20**	29		45	**32**	101
12		29	**25**	33		42	**34**	127
24		46	**34**	43		53	**42**	120

PermSort – One

Elapsed Time (s)

	DFS	*splitAll*1	*dfsBag*	*fdfsBag*	*BFS*	*bfsParallel*2	*bfsBag*	*fairSearch*2
1	41.07	**38.91**	44.97	69.52	70.82	**70.53**	80.75	165.36
2		**27.58**	30.59	50.79		**57.91**	75.72	124.14
4		**15.55**	16.92	33.13		**40.32**	41.99	78.24
8		**9.74**	10.89	17.02		**31.36**	36.66	59.93
12		**8.22**	10.03	21.98		**32.46**	34.63	61.62
24		9.51	**9.25**	21.14		30.34	**28.95**	74.05

Max. Resident Set Size (MiB)

	DFS	*splitAll*1	*dfsBag*	*fdfsBag*	*BFS*	*bfsParallel*2	*bfsBag*	*fairSearch*2
1	125	**124**	125	521	**638**	643	693	5389
2		143	**132**	525		**570**	507	5224
4		**140**	146	440		**577**	522	4149
8		**128**	158	367		**581**	441	3995
12		**149**	159	333		**472**	384	3540
24		**157**	159	310		**534**	337	3245

PermSort – All

Elapsed Time (s)

	DFS	*splitAll*1	*dfsBag*	*fdfsBag*	*BFS*	*bfsParallel*2	*bfsBag*	*fairSearch*2
1	**43.20**	**43.20**	47.96	68.93	71.16	**70.72**	79.25	166.68
2		**27.85**	32.83	53.49		**58.71**	81.97	124.47
4		**16.30**	18.28	43.54		**40.60**	57.78	81.35
8		**10.43**	11.72	32.40		**31.37**	44.54	66.98
12		**9.84**	10.13	35.24		**32.15**	40.03	72.11
24		10.72	**10.06**	30.26		**30.41**	44.04	83.92

Max. Resident Set Size (MiB)

	DFS	*splitAll*1	*dfsBag*	*fdfsBag*	*BFS*	*bfsParallel*2	*bfsBag*	*fairSearch*2
1	125	**124**	125	520	**638**	644	694	5468
2		**141**	143	524		**574**	595	5247
4		159	**137**	442		551	**519**	4177
8		**136**	167	408		562	**485**	3641
12		176	**169**	341		529	**413**	3865
24		**149**	155	339		499	**357**	3104

Half – One

Elapsed Time (s)

	DFS	*splitAll*1	*dfsBag*	*fdfsBag*	*BFS*	*bfsParallel*2	*bfsBag*	*fairSearch*2
1	1.12	0.99	0.97	**0.79**	0.75	**0.72**	0.79	1.75
2		1.14	**0.55**	**0.55**		0.72	**0.54**	1.97
4		0.90	**0.35**	0.36		**0.37**	**0.37**	1.41
8		1.33	**0.25**	**0.25**		0.26	**0.24**	1.21
12		0.72	**0.20**	0.23		0.24	**0.22**	1.01
24		1.67	**0.19**	0.25		0.32	**0.26**	0.59

Max. Resident Set Size (MiB)

	DFS	*splitAll*1	*dfsBag*	*fdfsBag*	*BFS*	*bfsParallel*2	*bfsBag*	*fairSearch*2
1	7	7	7	**4**	5	**4**	**4**	29
2		48	**6**	**6**		**6**	**6**	29
4		63	**6**	**6**		**7**	**7**	30
8		228	**10**	**10**		**10**	**10**	31
12		165	**12**	**12**		**12**	**12**	33
24		302	**20**	21		**20**	21	35

Half – All

Elapsed Time (s)

	DFS	*splitAll*1	*dfsBag*	*fdfsBag*	*BFS*	*bfsParallel*2	*bfsBag*	*fairSearch*2
1	2.17	1.82	1.82	**1.51**	1.57	1.57	**1.49**	1.75
2		**1.02**	1.14	1.11		1.46	**1.16**	2.05
4		1.81	**0.72**	0.74		0.76	**0.74**	1.43
8		0.56	**0.46**	0.50		0.54	**0.50**	1.24
12		**0.38**	0.39	0.41		0.49	**0.41**	1.07
24		**0.22**	0.37	0.41		0.64	**0.43**	0.66

Max. Resident Set Size (MiB)

	DFS	*splitAll*1	*dfsBag*	*fdfsBag*	*BFS*	*bfsParallel*2	*bfsBag*	*fairSearch*2
1	11	10	11	**5**	6	**5**	**5**	28
2		47	**7**	**7**		**6**	7	29
4		137	**8**	**8**		**8**	**8**	29
8		176	**11**	**11**		**11**	**11**	30
12		50	**13**	**13**		**13**	**13**	31
24		42	**21**	22		**21**	**21**	36

Last – One

Elapsed Time (s)

	DFS	*splitAll*1	*dfsBag*	*fdfsBag*	*BFS*	*bfsParallel*2	*bfsBag*	*fairSearch*2
1	0.39	**0.36**	0.44	**0.36**	0.27	**0.26**	0.39	0.57
2		**0.61**	0.78	0.80		**0.56**	0.84	1.58
4		**0.72**	0.80	0.91		**0.62**	0.92	1.60
8		0.91	0.64	**0.6**		0.74	**0.66**	2.17
12		0.92	0.60	**0.57**		0.94	**0.70**	2.55
24		1.27	0.79	**0.72**		1.87	**0.87**	4.12

Max. Resident Set Size (MiB)

	DFS	*splitAll*1	*dfsBag*	*fdfsBag*	*BFS*	*bfsParallel*2	*bfsBag*	*fairSearch*2
1	37	**36**	61	21	**16**	**16**	21	102
2		31	24	**23**		**20**	23	122
4		106	**22**	23		24	**23**	120
8		206	**22**	**22**		29	**23**	118
12		157	**24**	26		32	**24**	127
24		164	**27**	33		44	**31**	116

Last – All

Elapsed Time (s)

	DFS	*splitAll*1	*dfsBag*	*fdfsBag*	*BFS*	*bfsParallel*2	*bfsBag*	*fairSearch*2
1	0.4	0.38	0.46	**0.37**	0.28	**0.27**	0.42	0.58
2		**0.60**	0.74	0.79		**0.54**	0.91	1.52
4		**0.68**	0.81	0.86		**0.61**	0.97	1.66
8		0.98	0.68	**0.60**		0.74	**0.67**	1.94
12		1.18	0.63	**0.58**		0.94	**0.70**	2.52
24		2.39	0.77	**0.71**		2.29	**0.87**	4.20

Max. Resident Set Size (MiB)

	DFS	*splitAll*1	*dfsBag*	*fdfsBag*	*BFS*	*bfsParallel*2	*bfsBag*	*fairSearch*2
1	38	37	62	**21**	17	**16**	22	103
2		34	26	**23**		**19**	24	120
4		109	23	**23**		**23**	24	112
8		311	22	**21**		29	**23**	114
12		350	**23**	26		32	**24**	120
24		521	**26**	32		44	**33**	101

EditSeq

Elapsed Time (s)

	DFS	*splitAll*1	*dfsBag*	*fdfsBag*	*BFS*	*bfsParallel*2	*bfsBag*	*fairSearch*2
1	31.18	29.95	**29.28**	32.08	34.87	**33.9**	34.08	42.27
2		22.01	**19.06**	21.25		27.64	**22.69**	29.21
4		11.62	**10.44**	11.66		14.8	**13**	17.27
8		**7.25**	7.77	7.88		9.96	**8.88**	12.37
12		7.18	**6.94**	7.35		8.24	**8.63**	11.63
24		**8.03**	8.6	8.35		8.91	**8.84**	17.11

Max. Resident Set Size (MiB)

	DFS	*splitAll*1	*dfsBag*	*fdfsBag*	*BFS*	*bfsParallel*2	*bfsBag*	*fairSearch*2
1	829	**777**	792	1019	1018	**1004**	1009	1940
2		837	**788**	961		965	**946**	1748
4		891	**837**	907		898	**880**	1673
8		861	**811**	878		919	**859**	1478
12		**831**	855	841		899	**869**	1534
24		856	**829**	866		928	**843**	1385

NDNums

Elapsed Time (s)

	BFS	*bfsParallel2*	*bfsBag*	*fairSearch2*
1	**0.80**	0.81	0.96	oom
2		1.20	**0.52**	oom
4		1.35	**0.38**	oom
8		1.32	**0.28**	oom
12		1.35	**0.17**	oom
24		1.14	**0.07**	oom

Max. Resident Set Size (MiB)

	BFS	*bfsParallel2*	*bfsBag*	*fairSearch2*
1	**79**	80	90	oom
2		143	**61**	oom
4		266	**79**	oom
8		359	**84**	oom
12		382	**56**	oom
24		273	**30**	oom

C. Contents of the Data Medium

For further research this thesis includes a data medium with the complete material:

- Complete Benchmark Results Including Diagrams
- Code of the Bag of Tasks Implementations
- Version of KiCS2 with Parallel Search Strategies