Parallelizing DPLL in Haskell

Till Berger and David Sabel

Computer Science Institute
Goethe-University Frankfurt am Main
till.berger@stud.uni-frankfurt.de
sabel@ki.informatik.uni-frankfurt.de

Abstract: We report on a case study of implementing parallel variants of the Davis-Putnam-Logemann-Loveland algorithm for solving the SAT problem of propositional formulas in the functional programming language Haskell. We explore several state of the art programming techniques for parallel and concurrent programming in Haskell and provide the corresponding implementations. Based on our experimental results, we compare several approaches and implementations.

1 Introduction

Due to the ongoing developments in hardware, multiprocessor and multicore programming is becoming more and more popular. To benefit from this development, on the one hand it is necessary to parallelize known algorithms by modifying the existing sequential algorithms, and on the other hand (in the best case easy to use) programming primitives for parallel and/or concurrent programming have to be developed and evaluated.

For the functional programming language Haskell [Mar10] several approaches for parallel and concurrent programming exist (for overviews see [PS09, Mar12]). Our motivation of this paper is to evaluate the possibilities for parallel and concurrent programming by implementing a typical use case for parallelization in several variants. As a result we compare the different implementations experimentally by measuring their performance.

As an easy to parallelize but interesting problem we chose SAT solving, i. e. the problem of answering the question whether or not a propositional formula is satisfiable. More concretely, as the existing sequential algorithm we used the Davis-Putnam-Loveland-Logemann (DPLL) procedure [DP60, DLL62] which decides satisfiability of propositional formula in clause normal form, by using unit propagation and case distinction.

There are several investigations of parallelizing this algorithm (see e. g. [BS96, ZBH96, HJS09]) which is an ongoing research topic. However, our goal is not to provide a very fast implementation of DPLL, but to compare, explain, and investigate several possibilities for parallelization in the functional programming language Haskell and how they can be adapted to this particular use case.

A similar approach has been undertaken in [RF09] where different strategies for paral-

lelizing a nondeterministic search in Haskell were implemented and analyzed. The DPLL procedure was used as an example application. Compared to their approach we also provide implementations using (implicit) futures, a "parallel and" (which behaves like an amb operator), and we use the Eval monad, which supersedes Haskells's older `par` combinator and appeared after the work of [RF09]. As a further difference, our performance tests include satisfiable as well as unsatisfiable formulas.

Outline of the paper In Section 2 we briefly describe the satisfiability problem and the (sequential) DPLL algorithm which solves this problem. We also will provide a simple implementation in Haskell. In Section 3 we present our implementations of parallel variants of DPLL, using several programming libraries available for parallel and concurrent programming in Haskell. We will also briefly explain the corresponding libraries. In Section 4 we present and interpret our experimental results. Finally, in Section 5 we conclude and list some open questions left for further research.

More details of the undertaken case study (including more test results and further motivation) can be found in [Ber12].

2 The DPLL Algorithm

The DPLL algorithm [DP60, DLL62] takes a propositional formula in conjunctive normal form (i. e. a set of clauses) as input and decides whether or not the formula is satisfiable. It uses unit propagation (i. e. the combination of unit resolution and subsumption) and case distinction as techniques. DPLL is the core of many modern SAT solvers like Chaff [zCh12, MMZ$^+$01] and zChaff [zCh12], GRASP [SS96] or MiniSat [Min12, ES03].

We introduce some notation. A propositional *atom* is a propositional variable (e. g. x). A *literal* is an atom x (called a *positive literal*) or a negated atom $\neg x$ (called a *negative literal*). We will use l, l_i for literals. With $\bar{l}$ we denote $\neg x$ if l is a positive literal x and x if l is a negative literal $\neg x$. A propositional formula is in *conjunctive normal form* iff it is of the form $(l_{1,1} \vee \ldots \vee l_{1,m_1}) \wedge \ldots \wedge (l_{n,1} \vee \ldots \vee l_{n,m_n})$. We use clause sets instead of conjunctive normal forms, i. e. we write $\{\{l_{1,1}, \ldots, l_{1,m_1}\}, \ldots \{l_{n,1}, \ldots, l_{n,m_n}\}\}$, where the set $\{l_{i,1}, \ldots, l_{i,m_i}\}$ is called a *clause*. Clauses of the form $\{l\}$ are called *unit-clauses*. For a clause set $\mathcal{C}$, a literal l is *pure* in $\mathcal{C}$ iff $\bar{l}$ does not occur in $\mathcal{C}$.

In its core the DPLL procedure can be described by Algorithm 2.1 shown in Figure 1. The shown DPLL algorithm only decides satisfiability, but it is easy to adapt it to also generate models for satisfiable formulas by setting the literal l of unit clauses $\{l\}$ used for unit propagation in line 3 to true and setting the pure literals l chosen in line 7 to true.

In practice, further improvements are used to speed up search by using so-called conflict-driven backjumping instead of backtracking and learning clauses (see e. g. [NOT06] for an overview). However, for our case study we wanted to keep the core algorithm simple, so we did not use these improvements. Hence, our implementation of the DPLL algorithm in Haskell (which additionally computes a model for satisfiable clause sets) is very close to

Algorithm 2.1.

Input: *A propositional clause set* $\mathcal{C}$
Output: `true` *(*$\mathcal{C}$ *is unsatisfiable) or* `false` *(*$\mathcal{C}$ *is satisfiable)*

$DPLL(\mathcal{C})$:
(1) **if** $\emptyset \in \mathcal{C}$ **then return** `true`;
(2) **if** $\mathcal{C}$ = *emptyset* **then return** `false`;
(3) **if** $\exists$ *unit-clause* $\{l\} \in \mathcal{C}$ **then**
(4) $\mathcal{C}_1$:= *remove all clauses in* $\mathcal{C}$ *that contain literal* l;
(5) $\mathcal{C}_2$:= *remove all literals* $\overline{l}$ *occuring in* $\mathcal{C}_1$;
(6) **return** $DPLL(\mathcal{C}_2)$;
(7) **if** $\exists$ *pure literal* l *in* $\mathcal{C}$ **then**
(8) $\mathcal{C}_1$:= *remove all clauses in* $\mathcal{C}$ *that contain literal* l;
(9) **return** $DPLL(\mathcal{C}_1)$;
(10) *Choose a variable* x *that occurs in* $\mathcal{C}$;
(11) **return** $DPLL(\mathcal{C} \cup \{\{x\}\}) \wedge DPLL(\mathcal{C} \cup \{\{\neg x\}\})$;

Figure 1: The DPLL procedure

the pseudo code of Algorithm 2.1. Leaving out some helper functions, the implementation in Haskell is shown in Fig. 2. The functions not shown here are `findUnit` which searches for unit clauses, `resolve` which performs unit propagation, and `findLiteral` to find a next decision literal used in the case distinction. Literals are represented by integers where negative numbers represent negative literals. Compared to the pseudo algorithm our implementation does not perform deletion of isolated literals since searching and deletion is very expensive, and the run time gets slower if deletion is included.

3 Parallelizing the DPLL Algorithm in Haskell

Due to its tree-like recursion (the case distinction in line 11 of Algorithm 2.1), the DPLL algorithm is obviously parallelizable by evaluating both recursive calls in parallel. This can be seen if we look at the execution graph of the sequential algorithm. Figure 3 shows one such graph for our Haskell implementation executed on the example clause set

$$\{\{p,r\},\{p,s\},\{p,\neg r,\neg s\},\{\neg p,q,r\},\{\neg p,q,s\},\\ \{\neg p,q,\neg r,\neg s\},\{\neg q,r\},\{\neg q,s\},\{\neg q,\neg r,\neg s\}\}\ .$$

A node shows the selected literal in each step where the root node is the first step. If a node has only one child node, then it is an execution of unit propagation. Whereas if it has two child nodes, then it is an execution of the last rule. In this case, the left edge represents the positive path (i. e. the literal was assigned `true`), and the right edge represents the negative path (i. e. the literal was assigned `false`). Leaf nodes have either of the values `true` or `false`. For the example all leaves are `true`, and thus the formula is unsatisfiable. If the tree contains a path ending in a leaf marked with `false`, then a (partial) model of

```
type Literal   = Int
type Clause    = [Literal]
type Model     = [Literal]
type ClauseSet = [Clause]

dpll :: ClauseSet → Model → Model
dpll [] model = model
dpll clauses model
 | [] ∈ clauses = []
 | otherwise =
    case findUnit clauses of
      Just u → dpll (resolve u clauses) (u:model)
      Nothing →
        let dlit         = findLiteral clauses
            positivePath = dpll (resolve dlit clauses) (dlit:model)
            negativePath = dpll (resolve (- dlit) clauses) ((- dlit):model)
        in case positivePath of
            [] → negativePath
            xs → xs
```

Figure 2: The DPLL algorithm in Haskell

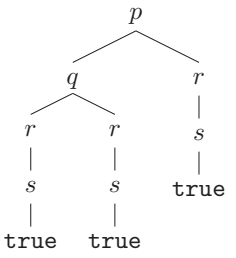

Figure 3: Example decision tree

the formula (i. e. a truth assignment) can be read off the corresponding path.

While sequential DPLL algorithms usually traverse the decision tree by a depth-first search, the idea for parallelization is to traverse all paths in parallel. Of course, this is a form of speculative parallelism, since perhaps unnecessary computations may be performed. On the other hand, even if we execute the parallel algorithm on a single processor (i. e. a concurrent evaluation which traverses the different paths in an interleaved manner), this may speed up the search by finding a model earlier.

Parallelization of the search with multiple processors is beneficial if all (or many) paths need to be searched (e. g. if the input clause set is unsatisfiable), or if a satisfying assignment is on a path that would be found late by the sequential search.

The parallel variant of the DPLL algorithm requires to synchronize the results of the com-

putations along different paths. For speeding up search, the algorithm should stop as early as possible. However, in the pure functional setting this requires to use a "parallel and" for synchronizing the results (i. e. an "and" operator which is non-strict in both of its arguments). Unfortunately, some libraries for parallel programming available for Haskell do not provide such a mechanism, and thus for several of our implementations we will use a "sequential and". However, in the impure world (using Haskell's `IO` monad), we will also provide an implementation that behaves like a "parallel and". As opposed to a "parallel and", a "sequential and" cannot achieve a speedup if the two paths traversed in parallel are both satisfiable, and one of them can be computed significantly faster than the other one.

Now, if we fork off computations at every decision point, the overhead for managing the parallel computations might get bigger than the speedup through parallelization if our clause sets get very small. Additionally, the number of processors (or cores) is limited by the hardware. So it makes no sense to parallelize every decision point. Instead, we restrict the number of created parallel computations.

Detecting the point for stopping parallel execution and going back to the sequential algorithm is not trivial as the formula size e. g. does not shrink predictably, and the detection should not be too expensive. Otherwise we may just as well have found a solution already with the computational power used by the detection.

For our implementations we have chosen a considerably simple approach: a global search depth that is used as an additional parameter for DPLL. This is almost without cost but makes us need to adjust for a particular formula size or group to be of general use.

3.1 Implementation in Haskell

The general model of our parallel implementation of DPLL in Haskell is shown in Figure 4. There are only few modifications w. r. t. the sequential implementation: An additional parameter for restricting the parallelization depth is inserted, and for the case distinction either the parallel execution is performed, or if the bound is exceeded, sequential computation is used. Of course, the code for the parallel execution is still missing and will be filled with several variants which we will explain in the subsequent sections. Since we also use Concurrent Haskell which is inside Haskell's `IO` monad we use a second variant, which is analogous to the program frame shown, but uses monadic `do` notation.

There are several methods for parallelizing code in Haskell. We have implemented some variants of the DPLL algorithm using the methods available within the Glasgow Haskell Compiler (GHC) and also using some others available by libraries. In particular, we used the Eval monad (on top of which evaluation strategies are built) [THLP98, MML+10], Concurrent Haskell [PGF96, Pey01, PS09], and its extension with futures [SSS11]. We also considered the Par monad [MNP11], which is available as a separate package (*monad-par*). We will discuss at the end of this section why we did not include an implementation using it in our tests.

In the following we are going for a mixed approach at explaining Haskell's parallelization capabilities and our respective implementations: We are discussing them at the same time

```
type Threshold = Int

dpllPar :: Threshold → ClauseSet → Model → Model
dpllPar _ []      model = model
dpllPar i clauses model
 | [] ∈ clauses = []
 | otherwise =
    case findUnit clauses of
     Just u → dpllPar i (resolve u clauses) (u:model)
     Nothing →
      let
       dlit         = findLiteral clauses
       positivePath = dpllPar (i-1) (resolve dlit clauses) (dlit:model)
       negativePath = dpllPar (i-1) (resolve (- dlit) clauses) ((- dlit):model)
      in if i > 0 then
           ... -- parallelization
         else case positivePath of
                [] → negativePath
                xs → xs
```

Figure 4: Program frame for all pure parallel implementations

and only explain the methods that are actually used. For a more thorough introduction into parallelism and concurrency within Haskell we refer to the tutorial [Mar12].

3.1.1 Implementation using the Eval Monad

The Eval monad [MML+10] is the successor of the older parallelization API with `par` and `pseq` [THLP98]. It delivers a more implicit approach to parallelization: The programmer annotates possible subexpressions for parallelization (by using `rpar`) and the runtime system *may* execute the evaluation in parallel. Concretely, the runtime system manages a pool of so-called *sparks*, and evaluation of `rpar` adds a new spark to the pool. If resources are available (i. e. a processor core is unused), the runtime system takes the next spark of the pool and executes it. More precisely, every HEC (Haskell Execution Context), which exists roughly for every processor core [MPS09], has its own spark pool where expressions given to `rpar` are stored. Whenever a processor core has no further work to do, it first looks at its own spark pool before stealing from another. The oldest sparks are taken out first, so overall, they are roughly executed in order of their creation.

`rpar` is used within the monad and execution is initiated with `runEval`. To wait for the evaluation of an expression, `rseq` is used. One important fact to note is that by default `rpar` and `rseq` only evaluate to weak head normal form (WHNF). Evaluation to normal form can be forced with `rdeepseq`, which can be combined with `rparWith` to evaluate a parallel computation completely. These functions are all basic evaluation strategies. Evaluation strategies are an abstraction layer on top of the Eval monad. They allow to separate the algorithm from its evaluation. The function `parList` e. g. represents a strategy that

evaluates all elements of a list in parallel. For more information on evaluation strategies we refer to [MML+10].

For the parallelization part of the DPLL algorithm we use the Eval monad as follows:

```
runEval $ do x ← rpar negativePath
             return (case positivePath of
                       [] → x
                       xs → xs)
```

The negative path is annotated to be evaluated in parallel, and the program continues by first evaluating the positive path. Note that we do not need to use the strategy `rdeepseq`, which ensures that the normal form of an expression is evaluated. If we inspect the DPLL algorithm more closely, we can see that `negativePath` is always evaluated to normal form by `rpar`, because at the point where we know if the result list is empty or not (i. e. if a model exists or not), the expression has already been fully evaluated.

This implementation is subsequently called `EvalT`. As sparks are more lightweight than threads because they are just references to the respective expressions, we also implemented a variant with unbound parallelization depth, which we call `Eval`.

3.1.2 Implementations using Concurrent Haskell

Concurrent Haskell [PGF96, Pey01, PS09] extends Haskell's `IO` monad with concurrent threads. These can be spawned by the primitive `forkIO`, and they can be killed by using `killThread`. Concurrent Haskell provides so-called `MVars` for synchronization of and communication between threads. An `MVar` is either empty or filled. The primitive `newMVar` creates a new (filled) `MVar`. The operation `putMVar` tries to fill an empty `MVar`. If the `MVar` is already filled, the calling thread is blocked until some other thread empties the `MVar`. Reading the content of an `MVar` is done with `takeMVar` which reads the content and empties the `MVar`. Similar to `putMVar`, it blocks the reading thread if the `MVar` is already empty and waits until it gets filled. Threads that are blocked on an otherwise inaccessible MVar are automatically garbage collected by GHC's garbage collector.

Using Concurrent Haskell, we implemented several variants. The first few use the above mentioned "sequential and" for parallelization, so the alternative path is only checked for a solution after the main one has finished computing. With these implementations we also compare the use of implicit vs. explicit futures as explained below. The "parallel and", where the result of the faster path is taken first, is implemented in one variant.

The program frame for the implementations using Concurrent Haskell differs slightly from Figure 4 because they return their result in the `IO` monad.

3.1.2.1 Concurrent Futures A *future* [BH77, Hal85] is a variable whose value is initially not known, but becomes available in the future. The value of a concurrent future is computed by a concurrent thread (in Haskell by a monadic computation in the `IO` monad, [SSS11]). Note that for computing the value of a *pure* expression, inside such a future, a monadic action which evaluates the expression must be created, for instance by using

the primitive `evaluate`. Like lazy evaluation, other threads using the future can continue evaluation until there is some data dependency forcing the value of the future. One distinguishes between explicit and implicit futures. Explicit futures require a `force` command to request the value of a future, while for implicit futures this request is performed automatically by the underlying runtime system.

Explicit futures are easy to implement using Concurrent Haskell: The future is represented by an `MVar`, and the concurrent thread writes its result into this `MVar`. The `force` command simply reads the content of the `MVar`. Hence, explicit futures can be implemented as follows, where we additionally return the `ThreadId` of the thread created:

```
type EFuture a = MVar a
efuture :: IO a → IO (ThreadId,EFuture a)
efuture act = do ack ← newEmptyMVar
                 tid ← forkIO (act >>= putMVar ack)
                 return (tid,ack)
force :: EFuture a → IO a
force x = readMVar x
```

Implicit futures are not implementable in Concurrent Haskell, but they can be implemented by using the primitive `unsafeInterleaveIO`[1], which delays the computation of a monadic action. The implementation in Haskell is:

```
future :: IO a → IO (ThreadId,a)
future act = do ack ← newEmptyMVar
                tid ← forkIO (act >>= putMVar ack)
                result ← unsafeInterleaveIO (takeMVar ack)
                return (tid,result)
```

Analogous to explicit futures, an `MVar` is used to store the result of the concurrent computation, but the code for creation of the future already contains the code for reading and returning the whole result. However, this second part of the code is delayed by `unsafeInterleaveIO`, which means that only if some other thread demands the value of the future, the code is executed.

Although this implementation makes use of the unsafe operation `unsafeInterleaveIO`, in [SSS12] it was shown that this specific use is safe since this extension of Haskell (i. e. Concurrent Haskell with implicit futures) is a conservative extension[2].

Our implementations for a concurrent DPLL algorithm with explicit and implicit futures are `ConE` and `Con` respectively. The parallelization in `ConE` is implemented as follows:

```
do (tid, npvar) ← efuture negativePath
   pp ← positivePath
   case pp of
       [] → force npvar >>= return
       xs → killThread tid ≫ return xs
```

[1] which is not part of the Haskell standard, but available in all major implementations of Haskell

[2] The result in [SSS12] does not include killing of the thread, but should be extensible to this situation

`Con`, using an implicit future, looks very similar, but the result does not need to be explicitly forced, it can directly be used:

```
do (tid, np) ← future negativePath
   pp ← positivePath
   case pp of
       [] → return np
       xs → killThread tid ≫ return xs
```

While implementing `Con`, we also implemented a variant where the order of the first two lines is switched, which resulted in the variant called `Con′`:

```
do pp ← positivePath
   (tid, np) ← future negativePath
   case pp of
       [] → return np
       xs → killThread tid ≫ return xs
```

Even though one might expect that this variant sequentializes the whole search, this is not true as our results will show. Through the use of the implicit future, the evaluation of the negative path is not forced with `return np`, but on the last branching point before, with `case pp of`. Thus, after nested recursive execution, the action `pp ←positivePath` is not strictly executed before the future for the negative path is created. In other words, as soon as the positive path is seen to have no result, a "pointer" to the negative-path computation is returned (`return np`). At the last branching point before, the negative path can now be forked off before the results of the positive path are completely forced with `case pp of`. In summary, the execution of the variant `Con′` is like first walking sequentially along the leftmost path of the decision tree, and then forking bottom-up so that the negative paths of different levels in the search tree are computed in parallel.

3.1.2.2 Checking the faster path first Using the mechanics of `MVars`, we can check the results of the paths in the order their computation finishes by letting them both write to the same `MVar`. The slower thread is blocked until the result of the faster one is read and taken out of the `MVar`. Then it can write to the `MVar`, and we can get its result with a second read on the `MVar`. If the first one already returned a solution, we kill both threads – one of the two `killThread` operations simply does nothing.

```
do pvar  ← newEmptyMVar
   tidp  ← forkIO (positivePath ≫= putMVar pvar)
   tidn  ← forkIO (negativePath ≫= putMVar pvar)
   first ← takeMVar pvar
   case first of
       [] → takeMVar pvar ≫= return
       xs → killThread tidp ≫ killThread tidn ≫ return xs
```

We call this implementation `Amb` (since it models McCarthy's *amb* [McC63]). Depending on which path (thread) finishes first, the resulting model (if one exists) may be different.

3.1.3 The Par Monad

As stated earlier, we did consider the Par monad, but excluded our implementation using it from the tests. The decision was made because the Par monad does not support speculative parallelism. It allows to create concurrent threads, and organizes communication between them with variables of the type `IVar`, which works similar to Concurrent Haskell's `MVar`. But there is no way to cancel an ongoing concurrent computation, and unreachable threads are not garbage collected like in Concurrent Haskell. An implementation using the Par monad would only be useful for (nearly) completely unsatisfiable formulas. There is a modification of the Par monad – found in the blog entry [Pet11] – that enables cancellation of threads, but we did not investigate an implementation using this modified Par monad.

4 Experimental Results

We tested several parallel implementations of the DPLL algorithm together with the sequential one using the Criterion package [Cri12]. The set of input formulas is part of the SATLIB project [HS00], which provides randomly generated 3-SAT formulas. The tested clause sets consist of 125 or 150 propositional variables and 538 or 645 clauses. The clause sets are divided into satisfiable and unsatisfiable sets (these are named to make the parameters explicit: the names are either *uuf-xxx-yyy* or *uf-xxx-yyy* where *uuf* means *unsatisfiable formula*, and *uf* means *satisfiable formula*; *xxx* is the number of variables, and *yyy* is the number of clauses). For the satisfiable formulas we tested 20 formulas of each set, and for the unsatisfiable formulas we tested 10 formulas of each set.

All tests were performed on a system with two quad-core processors of type AMD Opteron 2356 and 16 GB main memory. Every test was repeated 20 times to obtain the average execution time. The code was compiled with GHC version 7.4.2, and all variants apart from the sequential one were run with the parallel garbage collection switched on (which is the default); disabling the parallel garbage collection reduced performance for all parallel variants, only the sequential one ran faster without. Like the parallel variants, the sequential variant was compiled with threading support switched on but the runtime system was restricted to one core on execution.

The time we compare all runtimes to is the runtime of the sequential variant, which we call `Seq`. Subsequently, all runtimes are displayed as relative numbers; the runtime of the sequential variant is defined as 1.0 for every tested formula. Thus, for the sequential variant, we omit the relative numbers, but display the absolute runtimes for orientation.

The complete set of results – together with the source code – can be found at

`http://www-stud.cs.uni-frankfurt.de/~tilber/davis-putnam`.

Some of our results are shown in Figures 5 and 6. We also measured the space usage (by GHC's statistic output). Figure 7 shows some of these results.

Implementation	Threshold # Cores	1 2	2	8	2 4	8	3 8	12	24
uf125-538									
`Seq`	Mean	1.17 sec.							
`Amb`	Mean	**0.75**	1.49	4.47	1.05	2.43	1.03	1.78	1.99
	Median	0.95	1.37	1.77	0.86	0.97	**0.59**	0.69	0.74
`Con`	Mean	1.19	1.95	4.59	**1.14**	2.48	1.28	1.86	2.02
	Median	1.19	1.69	1.96	1.01	1.11	1.04	**0.71**	0.74
`Con′`	Mean	1.11	1.05	0.97	1.12	0.72	1.08	0.62	**0.56**
	Median	1.09	1.09	0.87	1.12	0.64	1.11	0.49	**0.44**
`EvalT`	Mean	1.02	0.98	1.07	0.92	0.87	0.91	**0.82**	0.83
	Median	1.08	1.07	1.07	1.02	0.90	0.88	0.58	**0.57**
`Eval`	Mean	1.01			0.87		0.82		
	Median	1.07			0.89		**0.61**		
uf150-645									
`Seq`	Mean	5.27 sec.							
`Amb`	Mean	**0.51**	2.49	10.33	1.31	5.57	1.28	4.62	5.77
	Median	0.41	0.78	1.59	0.43	0.86	**0.35**	0.64	0.77
`Con`	Mean	1.10	1.89	10.35	**1.03**	5.90	1.26	4.77	5.65
	Median	1.10	1.15	1.64	0.91	0.89	0.77	**0.65**	0.78
`Con′`	Mean	1.10	1.09	0.86	1.13	0.69	1.02	0.55	**0.48**
	Median	1.10	1.09	0.74	1.11	0.59	1.05	0.45	**0.37**
`EvalT`	Mean	1.00	1.08	0.96	0.89	0.78	0.95	0.74	**0.73**
	Median	1.07	1.03	0.98	0.80	0.69	0.68	0.48	**0.46**
`Eval`	Mean	0.94			0.78		**0.77**		
	Median	0.98			0.63		**0.47**		

Figure 5: Test results for satifisiable formulas (runtimes relative to `Seq`)

Implicit and Explicit Futures We did not include runtimes for the implementation with explicit futures (`ConE`) since they are almost the same as the runtimes for the implicit-future variant (`Con`). The reason is that the `force` command in the variant with explicit futures is almost exactly at the point where the result is needed. The only difference is that in the implicit-future variant, the negative path computation does not need to be evaluated before it is handed to the parent branching point. This difference is only essential for `Con′` where the negative path is forked off before the evaluation of the positive path is completely forced with `case pp of`.

Parallelization Depth for the Implementations using the Eval Monad Our results show that for using the Eval monad, the implementation with a depth bound (`EvalT`) performs not as good as the implementation without any bound (`Eval`). The numbers which are slightly higher for `Eval` can be attributed to measuring inaccuracy. In general,

Implementation	Threshold	1	2	8	2	8	3	12	24
	# Cores	2			4		8		
uuf125-538									
`Seq`	Mean	3.52 sec.							
`Amb`	Mean	0.75	0.70	0.57	0.59	0.33	0.47	0.21	**0.21**
	Median	0.72	0.69	0.57	0.59	0.33	0.45	0.21	**0.21**
`Con`	Mean	0.75	0.79	0.60	0.58	0.34	0.48	0.22	**0.22**
	Median	0.75	0.77	0.60	0.57	0.34	0.47	0.22	**0.22**
`Con′`	Mean	1.28	1.03	0.63	1.11	0.51	1.01	0.41	**0.36**
	Median	1.29	1.05	0.62	1.15	0.48	0.99	0.39	**0.34**
`EvalT`	Mean	0.68	0.66	0.55	0.51	0.31	0.41	0.19	**0.19**
	Median	0.66	0.61	0.54	0.50	0.31	0.41	0.19	**0.18**
`Eval`	Mean	0.54			0.30		**0.19**		
	Median	0.54			0.30		**0.18**		
uuf150-645									
`Seq`	Mean	10.80 sec.							
`Amb`	Mean	0.83	0.74	0.57	0.65	0.32	0.41	**0.20**	0.21
	Median	0.73	0.72	0.57	0.62	0.32	0.41	**0.20**	0.21
`Con`	Mean	0.87	0.82	0.59	0.69	0.33	0.43	**0.20**	0.21
	Median	0.79	0.78	0.59	0.70	0.33	0.42	**0.20**	0.21
`Con′`	Mean	1.31	0.98	0.60	1.07	0.44	0.97	0.32	**0.27**
	Median	1.34	0.91	0.59	0.98	0.44	0.96	0.31	**0.26**
`EvalT`	Mean	0.71	0.65	0.55	0.54	0.30	0.36	0.18	**0.17**
	Median	0.66	0.65	0.54	0.54	0.30	0.36	0.18	**0.17**
`Eval`	Mean	0.54			0.29		**0.18**		
	Median	0.54			0.29		**0.18**		

Figure 6: Test results for unsatisfiable formulas (runtimes relative to `Seq`)

the higher the threshold for `EvalT`, the shorter the runtime. Only in some cases a small threshold is faster, but overall, `Eval` has the best mean behavior. For unsatisfiable formulas the advantage of `Eval` is even clearer. A smaller partitioning of the search space seems to speed up the search more than the overhead for managing more sparks slows it down.

Comparison of `Eval` and `Con` As variants `Eval` and `Con` share the same parallelization approach – although the first uses parallelism, the latter concurrency –, one might expect them to perform similar. But `Con` is noticeably more fragile regarding the parallelization depth than `EvalT`. The reason is that the overhead of creating many threads in Concurrent Haskell is too high compared to the overhead that managing sparks in the Eval monad creates. Measuring the space behavior of both variants, we can see that `Con` consumes much more space than `Eval`, and consequently requires also more time for garbage collection. A relatively small threshold seems thus to yield the best results. But even considering that,

Implementation	Threshold	1	2	4	8	16	32	100
uf125-538								
`Seq`	Mean	4.01 MB						
`Amb`	Mean	1.2	1.5	2.6	4	6.9	7	7
`Con`	Mean	1.2	1.5	2.5	4.5	7.3	7.2	7.3
`Con′`	Mean	1	1.1	1.2	1.3	1.5	1.5	1.5
`EvalT`	Mean	1.2	1.6	1.6	1.6	1.6	1.6	1.6
`Eval`	Mean	1.6						
uf150-645								
`Seq`	Mean	4.77 MB						
`Amb`	Mean	1.2	1.6	3.4	7.8	17	17.8	18.4
`Con`	Mean	1.2	1.6	3.3	9.2	19	19	19.3
`Con′`	Mean	1	1.1	1.2	1.5	1.6	1.7	1.7
`EvalT`	Mean	1.3	1.7	1.7	1.7	1.7	1.7	1.7
`Eval`	Mean	1.7						

Figure 7: Space behavior for satisfiable formulas on four cores (relative to `Seq`)

`Eval` performs consistently better as it does not have such extreme outliers as `Con` (which result in the very high average runtimes for higher thresholds).

Parallelization Depth for `Con′` The implementation `Con′` also performs almost consistently better than `Con` for satisfiable formulas; but in another way than `Eval`. A noticeable difference is that for `Con′` the parallelization depth needs to be relatively high until measurable parallelization resulting in speedup happens. This is because of the parallelization being executed bottom-up. Like for `EvalT`, the higher the threshold, the faster the execution. Besides, a threshold of `100` is enough to completely parallelize the algorithm for all tested formulas – the maximum branching depth lies at about 50.

This also holds for unsatisfiable formulas. For these, `Con′` (with high threshold) is a bit slower than `Con` – but only on eight cores. This also indicates that parallelization happens later. As stated before, the positive path is evaluated only partly until a thread for the negative path is forked.

Inspecting the space behavior shows that `Con` consumes much more space than `Con′` for satisfiable formulas. As a consequence `Con` consumes more time for garbage collection than `Con′`. The reason might be that `Con′` creates a fewer number of threads, and the created threads are often necessary for the result of the computation.

Parallelization Depth for `Amb` The most ambiguous variant is `Amb`. It is even more fragile than `Con` regarding the parallelization depth. For satisfiable formulas a small depth is in most cases best so that the number of threads created equals the number of processor cores used. The mean is always worse for a higher threshold – meaning that there are

some extreme outliers that get worse the higher the threshold is. But as the median shows, a higher threshold yields a better runtime for some formulas. There are some extremely fast runtimes – only about 5 % of the sequential runtime – but that only happens on two cores with minimal threshold. Using more cores, the best single runtimes lie at about 18 %, and they are fewer. With a higher threshold, the time needed for garbage collection gets too high in most cases because of the high number of threads created. The increasing space usage can be seen in Figure 7.

For unsatisfiable formulas the almost reverse holds: Here, a threshold as high as possible is preferable. Though in some cases if it gets too high, the runtime slightly increases again. In this case, the thread-managing overhead seems to outweigh the better partitioning of the search space.

Comparing all Implementations Regarding unsatisfiable formulas, `Eval` performs best. But all parallel variants yield a noticeable advantage over the sequential one in this case. The relatively simple implementations perform very well when the whole decision tree needs to be traversed, and no speculative parallelism is actually performed.

For the satisfiable formulas, results are mixed. `Amb` is the fastest in some single cases where the advantage that it first checks the faster of both paths, translates to a very short runtime. But in many other cases the runtimes are extremely bad. This can be seen from the harsh difference between mean and median values for `Amb`. In these cases, the parallelization overhead seems to be too high. This is probably due to the fact that even more threads than with `Con` are created, which is also indicated by the fact that a high depth bound increases the runtimes even more than for `Con`. `Eval` is more stable but also has some outliers, which are less extreme. Its average results are thus looking better but, considering the number of cores utilized, not very well with only about 75 % of the sequential runtime for the medium sized formulas. In the average, `Con′` performs best for satisfiable formulas. Compared to the other two variants, runtimes are very stable resulting in mean values that are only slightly larger than the medians.

For `Eval`, the runtime variations were more noticeable with eight cores. Using less cores, the mean values are only a bit higher, whereas the median values improve with more cores. This suggests that the coordination costs get much higher in some cases when using more cores. Although the results for unsatisfiable formulas suggest that sparks behave indeed very cheap and that spark handling of the threaded GHC runtime system performs well, the results for satisfiable formulas show that there are cases where spark handling creates a noticeable overhead (though less much so than concurrent threads).

The depth bound approach is probably problematic if the search tree of a particular formula is very unbalanced. In that case the depth may in the worst case either be too big so that we do not stop parallelizing soon enough in the faster to solve path. This way, we lose time because of management overhead. Or it may be too low so that we lose parallelization potential in one path.

5 Conclusion

Concluding, our results show that the GHC runtime system performs very well when parallelizing sequential algorithms without speculative parallelism – in our case when unsatisfiable formulas are tested. Here, the most implicit approach, the Eval monad, yields the best results due to very little overhead. In case speculative parallelism is actually utilized when testing satisfiable formulas, our depth bound approach seems to be too simple though `Con′` performs surprisingly well compared to the other variants. For better results on satisfiable formulas, other techniques for bounding the parallelization depth – like trying to estimate the workload for a (partly reduced) formula – could be applied in further research. Furthermore, it might be interesting to see if a bottom-up forking like it happens with `Con′` is still advantageous in a more space-optimized implementation.

Acknowledgments We thank Manfred Schmidt-Schauß for reading this paper and for comments on this paper. We also thank the anonymous reviewers for their valuable comments.

References

[Ber12] T. Berger. Entwurf und Implementierung paralleler Varianten des Davis-Putnam-Algorithmus zum Erfüllbarkeitstest aussagenlogischer Formeln in der funktionalen Programmiersprache Haskell. Bachelorarbeit, Goethe-University Frankfurt am Main, Germany, 2012.

[BH77] H. C. Baker, Jr. and C. Hewitt. The incremental garbage collection of processes. In *Proc. Artif. intell. and prog. lang.*, pages 55–59. ACM, 1977.

[BS96] M. Böhm and E. Speckenmeyer. A Fast Parallel SAT-Solver - Efficient Workload Balancing. *Ann. Math. Artif. Intell.*, 17(3-4):381–400, 1996.

[Cri12] Criterion. Homepage, 2012. http://hackage.haskell.org/package/criterion.

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

[DP60] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *J. ACM*, 7(3):201–215, 1960.

[ES03] N. Eén and N. Sörensson. An Extensible SAT-solver. In *Proc. SAT 2003.*, volume 2919 of *Lecture Notes in Comput. Sci.*, pages 502–518. Springer, 2003.

[Hal85] R. H. Halstead, Jr. MULTILISP: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7:501–538, 1985.

[HJS09] Y. Hamadi, S. Jabbour, and L. Sais. ManySAT: a Parallel SAT Solver. *JSAT*, 6(4):245–262, 2009.

[HS00] H. H. Hoos and T. Stützle. SATLIB: An Online Resource for Research on SAT. In *SAT2000: Highlights of Satisfiability Research in the year 2000*, Frontiers in Artificial Intelligence and Applications, pages 283–292. Kluwer Academic, 2000.

[Mar10] S. Marlow. Haskell 2010 Language Report. `http://www.haskell.org/`, 2010.

[Mar12] S. Marlow. Parallel and Concurrent Programming in Haskell. In *CEFP 2011*, volume 7241 of *Lecture Notes in Comput. Sci.*, pages 339–401. Springer, 2012.

[McC63] J. McCarthy. A Basis for a Mathematical Theory of Computation. In *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.

[Min12] MiniSat. Homepage, 2012. http://minisat.se/.

[MML+10] S. Marlow, P. Maier, H.-W. Loidl, M. Aswad, and P. W. Trinder. Seq no more: better strategies for parallel Haskell. In *Proc. Haskell 2010*, pages 91–102. ACM, 2010.

[MMZ+01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proc. 38th DAC*, pages 530–535. ACM, 2001.

[MNP11] S. Marlow, R. Newton, and S. L. Peyton Jones. A monad for deterministic parallelism. In *Proc. Haskell 2011*, pages 71–82. ACM, 2011.

[MPS09] S. Marlow, S.L. Peyton Jones, and S. Singh. Runtime support for multicore Haskell. In *Proc. ICFP 2009*, pages 65–78. ACM, 2009.

[NOT06] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(*T*). *J. ACM*, 53(6):937–977, 2006.

[Pet11] T. Petricek. Explicit speculative parallelism for Haskell's Par monad. http://tomasp.net/blog/speculative-par-monad.aspx, 2011.

[Pey01] S. Peyton Jones. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction*, pages 47–96. IOS-Press, 2001.

[PGF96] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *23th POPL*, pages 295–308. ACM, 1996.

[PS09] S. Peyton Jones and S. Singh. A tutorial on parallel and concurrent programming in Haskell. In *6th AFP*, pages 267–305. Springer, 2009.

[RF09] F. Reck and S. Fischer. Towards a Parallel Search for Solutions of Non-deterministic Computations. In *Proc. ATPS 2009*, volume 154 of *LNI*, pages 2889–2900. GI, 2009.

[SS96] J. P. M. Silva and K. A. Sakallah. GRASP – a new search algorithm for satisfiability. In *Proc. ICCAD '96*, pages 220–227. IEEE Computer Society, 1996.

[SSS11] D. Sabel and M. Schmidt-Schauß. A contextual semantics for Concurrent Haskell with futures. In *13th PPDP*, pages 101–112. ACM, 2011.

[SSS12] D. Sabel and M. Schmidt-Schauß. Conservative Concurrency in Haskell. In *LICS*, pages 561–570. IEEE, 2012.

[THLP98] P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithms + Strategy = Parallelism. *J. Funct. Program.*, 8(1):23–60, 1998.

[ZBH96] H. Zhang, M.P. Bonacina, and J. Hsiang. PSATO: a Distributed Propositional Prover and its Application to Quasigroup Problems. *J. Symb. Comput.*, 21(4):543–560, 1996.

[zCh12] zChaff. Homepage, 2012. http://www.princeton.edu/∼chaff/zchaff.html.