Chapter 6

Parallel Heuristic Search in Haskell

Michelle Cope, Ian Gent[1] and Kevin Hammond[1]

Abstract: Parallel heuristic search algorithms are widely used in artificial intelligence. This paper describes novel parallel variants of two standard sequential search algorithms, the standard Davis Putnam algorithm (DP); and the same algorithm extended with conflict-directed backjumping (CBJ). Encouraging preliminary results for the GpH parallel dialect of the non-strict functional programming language Haskell suggest that modest real speedup can be achieved for the most interesting hard search cases.

6.1 INTRODUCTION

The aim of this work is to investigate parallel implementations of heuristic search algorithms. The algorithms used are based on the Davis Putnam algorithm [7, 6, 12] optionally extended with conflict-directed backjumping (CBJ) [21, 2, 3]. A typical application of these algorithms is to solve propositional satisfiability (SAT) problems. SAT is extremely valuable to artificial intelligence because of its direct relationship with deductive reasoning. Many other problems in artificial intelligence can be encoded in SAT, including default reasoning, diagnosis, image interpretation [22] and constraint satisfaction (CSP) problems. SAT is thus regarded as fundamental to solving many artificial intelligence problems and much effort has been spent in developing efficient algorithms to solve SAT-related problems (e.g. [29]).

In order to solve a SAT problem, the Davis Putnam algorithm attempts to generate a set of assignments to the free variables (or *literals*) of the input proposition under which that proposition is *true*. The conflict-directed backjumping (CBJ)

[1]School of Computer Science, University of St Andrews, North Haugh, St Andrews, KY16 9SS, Scotland; Fax: +44 (0)1334-463278; Email:
`{ipg,kh}@dcs.st-and.ac.uk`

algorithm additionally generates *conflict sets* comprising assignments to subsets of those literals under which the proposition must be *false*. These conflict sets are used to prune the search space by eliminating areas where prior information shows that there can be no solution. Clearly, both algorithms are potentially highly parallel in that different sets of assignments may be searched independently. Use of conflict sets and other techniques aimed at optimising the sequential algorithms, such as shared data structures, may, however, introduce either accidental or essential sequential dependencies. This is especially so in the case of CBJ, because the set of assignments to be searched in one branch may depend on the conflict set returned from another. In its normal presentation CBJ is thus essentially sequential.

Rather than aiming for an optimal sequential algorithm, as is usual in the literature, we rather aim to find an algorithm that is amenable to parallel implementation, even if it is sequentially sub-optimal. All the algorithms described here were coded in Haskell [20], which allows rapid prototyping of parallel solutions using the GpH (Glasgow parallel Haskell) dialect [25]. Development of parallel algorithms is supported by the HasPar [16] suite of performance monitoring tools, which incorporates the GranSim simulator [15] plus good performance visualisation tools. Both sequential and parallel algorithms are described in detail in Section 6.2.

Each of the algorithms was run on a number of artificially generated SAT instances using a random generator based on the fixed-clausal length model [9], and taking specific instances of search problems from the DIMACS database [8], as described in Section 6.3.

6.2 HEURISTIC SEARCH ALGORITHMS FOR SAT

All the algorithms introduced here have a similar structure: they each assess a set of clauses to determine the existence or absence of a *satisfying* set of literal assignments. The input clausal set is in *conjunctive normal form* (CNF): a conjunction of clauses where each clause is a disjunction of literals. Assuming propositional logic, a clausal set is *satisfiable* if at least one literal in every clause is assigned to *true*, whereupon the value of each clause is also *true*. For example, $(x \vee y) \wedge (x \vee z)$ is *satisfiable* under the two sets of literal assignments $\{\ x\ = true\ \}$ and $\{\ y\ = true,\ z\ = true\ \}$.

6.2.1 The Davis-Putnam Algorithm (DP)

Most solution strategies for SAT can be classified either as *stochastic* or *systematic*. *Stochastic* or *local search* algorithms, such as WSAT [24] or GSAT [23], explore a search space randomly by making local alterations to a working assignment without memory of past assignment attempts. These algorithms are *incomplete*: they are not guaranteed to find a solution if one exists, and thus cannot declare that an instance is insoluble if they do not find a solution. Reviews of stochastic algorithms can be found in [11, 10].

procedure DP(Σ)

(Sat)	**if** $\Sigma = \{\}$ **then return** *satisfiable*
(Empty)	**if** Σ contains an empty clause **then return** *unsatisfiable*
(Tautology)	**if** Σ contains a tautologous clause, c **then return** DP(Σ - {c})
(Unit propagation)	**if** Σ contains a unit clause, $\{l\}$ **then return** DP(Σ simplified by assigning l to *true*)
(Literal deletion)	**if** Σ contains a literal l but not $\neg l$ (the negation of l) **then return** DP(Σ simplified by assigning l to *true*)
(Split)	**if** DP(Σ simplified by assigning literal, l, to *true*) is *satisfiable* **then return** *satisfiable* **else return** DP(Σ simplified by assigning $\neg l$ to *true*)

FIGURE 6.1. The Davis-Putnam-Logemann-Loveland Algorithm, often called Davis-Putnam

Systematic or *global search* algorithms traverse the entire search space systematically checking all possible truth assignments for a given formula. They are *complete* given sufficient execution time, they will either find a solution if one exists or else will report a failure. The most popular systematic algorithms for SAT are variants of the Davis Putnam algorithm (DP), e.g. Tableau [5] or POSIT [10].

The original Davis Putnam algorithm uses a variable elimination or resolution rule which replaces the initial problem with a sub-problem that is usually larger and which, in general, requires exponential space to solve [7]. A later version due to to Davis, Logemann and Loveland replaces the resolution rule with a *splitting* rule that refines the original problem into two smaller sub-problems [6]. The latter algorithm is shown in Figure 6.1. The two versions are often confused. The later version (the one used here) is often referred to without qualification as the Davis Putnam algorithm.

6.2.2 Clausal Set Data Structure

The input clauses are structured as a list of pairs, where the first element represents a list of literals and the second is a clausal flag: **([Literal],Cflag)**. Each literal is an **(Int,LFlag)** pair, where the integer identifies the literal and the flag indicates its value.

```
type ClausalSet = [Clause]
```

```
type Clause  =    ([Literal], CFlag)

type Literal =    (Int, LFlag)
```

A literal may have one of three **Lflag** values: *true*; *false* – the literal has been removed as a result of unit subsumption (Section 6.2.3); or *indetermined* – a value is yet to be assigned.

The clausal flag indicates the state or value of the entire clause: *subsumed* – at least one literal in the clause is *true* and thus the entire clause is *true*; *empty* – all literal values are *false*, indicating that the clausal set is *unsatisfiable* under some set of literal assignments; or *unsubsumed* – the clausal value is yet to be determined.

```
data LFlag =     LTrue | LFalse | Indetermined

data CFlag =     Unsubsumed | Subsumed | Empty
```

The use of flags maintains all relevant information in a single data structure: the *clausal set*. An input clausal set has all clauses initially unsubsumed and the value of all literals are indetermined. All solutions, both sequential and parallel, manipulate the list of input clauses. On search completion, each solution returns a triple comprising an indication of whether the initial clausal set is satisfiable, the set of satisfying literal assignments if one exists, and the number of nodes visited (a performance metric). CBJ-based algorithms will also return a conflict set.

```
type SearchResult = (Boolean, Output_clausal_list, No_nodes)
```

6.2.3 Implementations of Davis-Putnam (DP)

Sequential Davis-Putnam (DP-Sequential)

Our implementation of DP-Sequential is based on the standard Davis Putnam algorithm [6] as shown in Figure 6.1, but excluding the tautology and pure literal deletion rules, in accordance with common practice. The clausal set will meet the requirements for precisely one of the following rules.

- **Empty Clausal Set**: If all clause flags are subsumed then the clausal set is *satisfiable*.
- **Empty Clause**: If the value of any clause flag is empty then the set is *unsatisfiable*.
- **Unit propagation Rule**: Any unsubsumed clause with only one remaining literal is *indetermined* is a *unit clause*. The single literal, l, in the unit clause is assigned to *true* and the clausal set is changed accordingly using subsumption and resolution as follows.
  - *Subsumption* sets the $CFlag$ to *subsumed* for all unsubsumed clauses which have l as a member.

– *Resolution* identifies those unsubsumed clauses which have the negation of l ($\neg l$) as a member. The value of $\neg l$ is changed to *removed*. Any clause with all literals effectively removed is denoted as *empty* otherwise its clausal flag remains unaltered.

The implementation of unit propagation need only identify the first unit clause in the list. Owing to the order in which the rules are implemented and the recursive nature of the algorithm all unit clauses are evaluated before the use of the splitting rule.

- **Splitting Rule**: In the general case, when no other rule applies, search continues by choosing a literal, l, according to some ordering heuristic. The heuristic we have implemented selects the first indetermined literal in the shortest unsubsumed clause. The search then continues by first finding any solutions with l set to *true* and then finding those with $\neg l$ set to *false*. After both branches have completed, the clausal set is then constructed appropriately using subsumption and resolution.

Parallel Davis Putnam (DP-parallel)

A major influence on the often large search times in DP-sequential is the use of left recursion in the splitting rule. The GpH `par` operator [17] can be used to indicate parallel evaluation of both left and right branches when the splitting rule is applied. DP-Parallel speculates that the search spaces on all right branches contribute to all solutions. Although, this naïve form of speculation does not prevent redundant search, it does search more of the problem space in less time than DP-Sequential. The code for DP-Sequential is modified by the introduction of a single call to `par` in order to evaluate the right branch in parallel with the rest of the search.

6.2.4 Conflict-dependent Backjumping (CBJ)

Sequential Conflict-dependent Backjumping (CBJ-Sequential)

The main deficiency of DP-Sequential is that it may explore the search space unnecessarily – both split branches are explored regardless of whether or not they could produce a solution. In contrast, CBJ uses *conflict sets* as a mechanism to identify redundant search spaces following backtracking. Such search spaces are then pruned without being searched.

A *conflict set* is a set of literals whose current assignment falsify the entire clausal set. We identify a conflict set whenever an empty clause is derived. The conflict set is the original set of literals in the clause that is now empty. Clearly this clause cannot be satisfied unless we change the value of one or more of these literals. Whenever a new conflict set is generated, the CBJ algorithm backtracks to reconsider the deepest literal, l, in the search tree. This literal is assigned the opposite value to its current assignment. Since, however, this new assignment might render another clause empty, a second conflict set may be generated and the algorithm may then backtrack further. When all possible truth values have been excluded for some literal, l', a contradiction is derived. To proceed further, the two conflict sets that have been generated by the positive and negative assignments to l' are resolved, effectively creating the union of both conflict sets and removing l'. The new conflict set is propagated up the search tree until no more contradictions occur.

Implementation of CBJ

The implementation of CBJ is similar to that of DP: the order of the rules are maintained but these rules are modified to handle conflict sets. The result of the search includes the conflict set that was produced.

```
type SearchResult = (Boolean, Output_clausal_list,
                     No_nodes, Conflict_Set)
```

- **Empty Clausal Set**: This is treated in the same way as for DP, except that an empty conflict set is also returned.
- **Empty Clause**: The presence of an empty clause indicates the existence of a conflict set. The conflict set is constructed by searching the clausal list for a clause marked empty.
- **Unit Propagation**: Unit propagation proceeds as in DP-Sequential unless the literal, u, in the unit clause is in a conflict set, say A. In this case, the literals of the original clause from which the unit clause is derived must have assignments which exclude any value for u. Consequently, the original clause constitutes a second conflict set, say B, which is resolved with A.
- **Splitting Rule**: The splitting rule is implemented similarly to DP-Sequential. Either or both branches may return a conflict set. The conflict sets are resolved and backtracking continues as previously described.

CBJ-Sequential does prevent some unnecessary search. However it suffers from the same deficiency as DP-Sequential: when splits occur the left branches are always searched sequentially before the right branches. The right branch at a split is thus not searched at all until the left branch reaches a base case: either a conflict set or a solution. Thus if the left branch detects a conflict set, it cannot be resolved until the right branch has completed. The Haskell implementation is given below.

```
conflictDirectedBackjumping :: [Clause] -> SearchResult
conflictDirectedBackjumping = cbj 1

cbj :: Int -> [Clause] -> SearchResult
cbj n xs
    | allSubsumed xs  = (True, xs, n, [])   -- Empty
Clausal Set
```

```
    | emptyClause xs  = (False, xs, n, findConflict xs)
    | otherwise       = cbj' n xs (unitClause xs)

cbj' n xs (Just w) = unitPropagation w n xs -- Unit Clause
cbj' n xs Nothing =                         -- Split
  let z = selectVariable xs
      a@(fa, _, na, ca) = cbj ( n + 1) (simplify   z  xs)
      b@(fb, _, _,  cb) = cbj (na + 1) (simplify (-z) xs)
  in
        if fa then a
        else if not $ isLitInConflict ca z then a
        else if fb then b
        else if not $ isLitInConflict cb z then b
        else (False, xs, (nb), resolve ca cb z)
```

Naïve Parallel CBJ (CBJ-Parallel)

The main deficiency of the sequential CBJ algorithm is the delay in resolving conflict sets. The parallel CBJ algorithm (CBJ-Parallel) is designed to avoid this weakness of the sequential CBJ solution. CBJ-Parallel implements mechanisms to detect and resolve areas of redundant search more quickly than the sequential solutions. Steps are also taken to reduce the creation of unnecessary threads. This algorithm is a naïve parallel version of sequential CBJ. Redundant search still occurs as no control is enforced over the speculation. The conflict-backjumping technique should, however, identify areas of redundant search sooner than DP and consequently we anticipate that CBJ-Parallel should outperform DP-Parallel. The code for CBJ-Sequential is modified as shown below by the introduction of a single call to `par`.

```
...
cbj' n xs Nothing =               -- Split
...
        b `par`                   -- Parallel Right Branch
         if fa then a
         else ...
```

Speculative Parallel CBJ (CBJ-Parallel-Spec)

The two parallel algorithms described above (DP-Parallel and CBJ-Parallel) take no deliberate measures to eliminate redundant search; neither algorithm exerts any control over the amount of speculation that is introduced. The reduction of redundant search is the central aim of the algorithm that is described in this section, CBJ-Parallel-Spec. Rapid resolution of conflict sets and control of speculation is achieved by using a wrapper mechanism and a fuel mechanism.

Each call to CBJ is embedded within a wrapper function. The wrapper inspects the state of a sub-area of the parent thread's search space before creating a new child thread that may deepen the search. Three possible states may be found:

No. of variables (nv)	No. of clauses (nc)	16 Processors (DP)	(CBJ)	32 Processors (DP)	(CBJ)	(CBJ-Spec)
20	200	1.45	1.50	1.48	1.51	1.46
20	90	1.57	1.60	1.58	1.60	1.58
40	200	4.70	4.48	4.82	4.66	4.52
40	90	0.96	0.96	0.96	0.96	0.92
50	200	1.92	2.06	2.80	2.22	2.69
50	90	1.00	1.00	1.00	1.00	1.00

FIGURE 6.2. Relative speedups for each algorithm on 16 & 32 processors

a solution to the problem, a conflict set, or a continuation for the remainder of the search. In the latter case, the parent is at a non-leaf node whereas in the former cases, the parent is at a leaf node. If a continuation is found, a child thread is created to compute the next level in the search tree, otherwise the state information is used to influence the overall result of the computation and to control the progress of the search.

In the trivial case, when a solution is found, the values of all executing threads can be ignored. The interesting case occurs when the existence of a conflict set has been detected. In this case, the conflict set is propagated to the level where it can be used to resolve the conflict, creating a continuation thread to explore the area of the search space in which the second conflict set may exist.

The throttling mechanism used here provides each function to be throttled with an extra parameter, an integer representing the thread's *fuel*. Each call to the function decrements the fuel. If the function produces a result, this is returned to the caller. Otherwise, when the fuel is decremented to 0, a pair is returned comprising an indication that the function call is incomplete plus a suspension representing the continuation of the call (with the fuel reset to its maximum value, `maxFuel`). The caller may choose to either continue evaluation with this continuation, or to take some other action as required by the algorithm.

6.3 PERFORMANCE RESULTS

The main factor affecting the performance of the sequential algorithms is the use of left recursion to force depth-first search. At a split, the right branch is not explored until the search space on the left branch has been explored. If a solution lies in the right branch but not in the left, then the production of the solution will be delayed. Parallel variants of CBJ should, in principle, be able to take advantage of speculation: for example, a small conflict set may be found on a right hand branch before a large conflict set is found on the left branch. The small set might then make the left hand branch redundant.

Relative speedup results for the parallel algorithms are shown in Figure 6.2. All the programs have been run on the same artificially generated instances. The average time cost is calculated for a sample of 50 SAT instances with different ratios of variables to clauses, representing different degrees of difficulty. Relative speedup is calculated as the ratio of the time taken on 1 processor (running the parallel version of the algorithm) to that taken on n processors. All performance results were obtained using the GranSim simulator, including base times for the sequential versions of the algorithm. Performance was measured in terms of simulated time steps ("clock ticks") for a high-latency distributed parallel machine approximating the St Andrews Beowulf cluster.

6.3.1 Analysis of Performance Results

Figure 6.2 shows an interesting disparity in performance between "easy" and "hard" instances of the SAT problem. We see little speedup on the easier instances. Hard instances occur where the ratio of clauses to variables are from 4 to 5, near the phase transition. We see the highest speedups on these problems, up to 4.8 at $nv = 40$ and 2.8 for $nv = 50$.

For all parallel algorithms, relative speedup between 0.92 and 4.8 is achieved using 16 processors, with an average speedup of 1.5. No significant improvement in relative speedup is evident when the number of processors is increased to 32. While these findings seem to concur with those of Gu [13], they are rather disappointing, and require further investigation. Unfortunately, no comparative performance results are available for fewer than 16 processors or for lower latency architectures. Such results might indicate whether the poor speedups are due to granularity, communication cost or are a consequence of the algorithm (for example, the use of a centralised data structure may have introduced excessive serialisation into the algorithm).

A comparison of the absolute execution times for each of the parallel algorithms suggests that CBJ-Parallel-Spec is slightly outperformed by the other two parallel algorithms, on average. Although the difference is small (1% – 4%), this finding was unexpected. A possible explanation is the cost of the wrapper mechanism that was used to control speculation. It is plausible that this mechanism as implemented introduces excess synchronisation overheads between split branches and that tuning will be necessary to reduce this overhead. Further work is needed to verify this conclusion and to determine the best way to avoid unnecessary synchronisation.

6.4 RELATED WORK

In practice, most sequential SAT algorithms can be mapped onto parallel computer systems, resulting in parallel SAT algorithms [12]. It may even be possible to achieve super-linear speedups if there are correlations between variable settings that lead to solutions. Parallel processing does not alter the worst-case complexity for SAT algorithms unless an exponential number of processors is available. However, parallel processing does delay the effect of exponential growth of computing time, allowing the solution larger SAT instances [12].

The sequential SAT algorithms described here depend strongly on efficient access to a centralised data structure, and thus the design of the data structure is fundamental to achieving good performance on a given computer architecture. Most early studies of CSP/SAT algorithms were performed on sequential machines [12]. However recent work has moved to parallel programming on multiprocessors. Bohm and Speckenmeyer have experimented with the parallelisation of the Davis Putnam algorithm on a message-passing Transputer system comprising 320 processors. For some small k each $2k$ processor solves a formula arising at depth k of a DP search tree, and computation ceases upon one processor finding a satisfiable assignment for its formula [4]. It was observed that the execution time was usually less than $\frac{N}{2k}$ where N was the time taken by the serial implementation.

Other parallel implementations of DP exist: for example, Zhang, Bonacina and Hsiang have implemented a simple master-slave distributed DP procedure and succeeded in exploiting parallelism without incurring the overhead of redundant search [28]. We are not aware, however, of prior work on the parallel implementation of DP with CBJ. In contrast, there do exist other parallel constraint-satisfaction problem algorithms (e.g. [13]) and Loidl has also implemented simple $\alpha - \beta$ search in GpH [18].

Gu et al. suggest that loosely-coupled multiprocessor computers offer limited performance improvements for solving SAT. Communication overhead between processors becomes a bottleneck as processors speed up. Their empirical research indicates that tightly-coupled parallel computing, which effectively reduces off-processor communication delays, is the key to the parallel processing of SAT formulae [12]. This work appears to be borne out by our central performance results, cited in Section 6.3.

6.5 SUMMARY AND CONCLUSIONS

This paper has presented a parallel implementation of the Davis Putnam algorithm (DP) for solving propositional satisfiability (SAT) problems, a fundamental problem in artificial intelligence. To our knowledge this is the first parallel algorithm that implements conflict-directed backjumping (CBJ) for DP. Overall, for the architecture that was simulated (a very high latency distributed machine), the parallel algorithms outperform the sequential algorithms on hard instances by a factor of 4-5 on either 16 or 32 processors, but perform no better than the sequential algorithms on easy instances. Further work is required to determine whether these results reflect limitations of the algorithm, of the architecture, or of the implementation.

While not spectacular, we are encouraged by these early results; the problem has wide application and is severely performance-limited. Achieving good parallel speedup would thus have important practical ramifications for the use of artificial intelligence techniques based on SAT or CSP. We are presently working on enhancing CBJ with discrepancy-based search using a continuation passing approach. We believe such an algorithm has the potential to yield improved parallelism over the approach described here.

REFERENCES

[1] R.J. Bayardo, and D.P. Miranker, “A Complexity Analysis of Space-Bounded Learning Algorithms for the Constraint Satisfaction Problem”. *Proc. 13th National Conference on Artificial Intelligence (AAAI-96)*, pp. 558–562, 1996.

[2] R.J. Bayardo Jr. and R.C. Schrag. Using CSP look-back techniques to solve exceptionally hard SAT instances. In *Principles and Practice of Constraint Programming – CP96*, pp. 46–60. Springer-Verlag, 1996.

[3] R.J. Bayardo Jr. and R.C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pp. 203–208, Menlo Park, CA, 1997. AAAI Press.

[4] M. Bohm and E. Speckenmeyer, “A fast parallel SAT-solver – efficient workload balancing”. *Proc. 3rd. International Symposium on AI and Mathematics*, 1994.

[5] J.M. Crawford, J.M and L.D. Auton, “Experimental Results on the Crossover Point in Random 3SAT”, *Artifical Intelligence* **81(1-2)**:31–57, 1996.

[6] M. Davis, G. Logemann, and D. Loveland, “A Machine Program for Theorem Proving”, *Comm. ACM* **5**:394–397, 1962.

[7] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. Association for Computing Machinery*, 7:201–215, 1960.

[8] DIMACS Challenge Committee. “The second DIMACS International Algorithm Implementation Challenge: General Information”.

[9] J. Franco and M. Paull, “Probabilistic analysis of the Davis Putnam procedure for solving the satisfiability problem”. *Discrete Applied Math.* **5**, pp. 77–87, 1983.

[10] J.W. Freeman, *Improvements to Propositional Satisfiability Search Algorithms*, PhD dissertation, Dept. of Computer and Information Science, University of Pennsylvania, 1995.

[11] I.P. Gent and T. Walsh, “The search for Satisfaction”, Internal Report, Dept. of Computer Science, University of Strathclyde, 1999.

[12] Gu, J., Purdon, P.W., Franco, J. and Wah, B.W. “Algorithms for the satisfiability problem: a survey”. *DIMACs Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, pp. 19–152, 1997.

[13] J. Gu and W. Wang, “A Novel Discrete Relaxation Architecture”. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, **14(8)**: 857–865, 1992.

[14] K. Hammond, C.V. Hall, H.-W. Loidl, and P.W. Trinder, “Parallel Cost Centre Profiling”, *Proc. 1997 Glasgow Workshop on Functional Programming*, Ullapool, Scotland, Sept. 1997.

[15] K. Hammond, H.-W. Loidl and A.S. Partridge, “Visualising Granularity in Parallel Programs: A Graphical Winnowing System for Haskell”. *Proc. HPFC’95 – Conf. on High Performance Functional Computing*, pp. 208–221, Denver, CO, April 10-12, 1995.

[16] K. Hammond, D.J. King, H.-W. Loidl and P.W. Trinder, “The HasPar Performance Evaluation Suite for GpH: a Parallel Non-Strict Functional Language”, Submitted to *Software – Practice & Experience*, 2000.

[17] K. Hammond and G. Michaelson (Eds.), *Research Directions in Parallel Functional Programming*, Springer-Verlag, 1999.

[18] H.-W. Loidl, *Granularity in Large-Scale Parallel Functional Programming*, PhD Thesis, Dept. of Comp. Sci., University of Glasgow, March, 1998.

[19] D. Mitchell, B. Selmen, and H. Levesque, H. "Hard and Easy Distributions of SAT problems". *Proc. Tenth National Conference on Artificial Intelligence (AAAI-92)*, San Jose, CA, July 1992, pp. 459–465.

[20] J.C. Peterson, K. Hammond (eds.), L. Augustsson, B. Boutel, F.W. Burton, J.H. Fasel, A.D. Gordon, R.J.M. Hughes, P. Hudak, T. Johnsson, M.P. Jones, S.L. Peyton Jones, A. Reid, and P.L. Wadler, *Report on the Non-Strict Functional Language, Haskell, Version 1.4*, 1997.

[21] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.

[22] R. Reiter and A. Mackworth,"A Logical Framework for Depiction and Image Interpretation", *Artificial Intelligence*, **42(2)**, pp. 125–155, 1989.

[23] B. Selman, D. Mitchell and H. Levesque, "A new method for Solving Hard Satisfiability Problems", *Proc. 10th. National Conf. on Artificial Intelligence*, pp 440–446, 1992.

[24] B. Selman, H. Kautz, and B. Cohen. Noise Strategies for Improving Local Search. In *Proceedings of the 12th National Conference on AI*, pp. 337–343. American Association for Artificial Intelligence, 1994.

[25] P.W. Trinder, K. Hammond, J.S. Mattson Jr., A.S. Partridge, and S.L. Peyton Jones, "GUM: A Portable Parallel Implementation of Haskell". *Proc. PLDI '96 – Programming Language Design and Implementation*, Philadelphia, PA, May 1996, pp. 78–88.

[26] P.W. Trinder, K. Hammond, H.-W. Loidl, and S.L. Peyton Jones, "Algorithm + Strategy = Parallelism", *Journal of Functional Programming*, **8**(1), January 1998.

[27] H. Zhang, "SATO: an Efficient Propositional Prover", *CADE 1997*, Springer LNCS 1249, pp. 272–275, 1997.

[28] H. Zhang, M.P. Bonacina and J. Hsiang, "PSATO: a distributed Propositional Prover and Its Application to Quasigroup Problems", *J. Symbolic Computation*, **21**, pp. 543–560, 1996.

[29] H. Zhang and M.E. Stickel, "Implementing the Davis-Putnam Algorithm by Tries", *Journal of Automated Reasoning*, **24**, pp. 277-296, 2000.

Contents