

Limited Redundancy Cooperative Search

LSIS Research Report Number 2002/001, February 2002

Laurent Henocque
laurent.henocque@lsis.org

Laboratoire des Sciences de l'Information et des Systèmes
LSIS (UMR CNRS 6168)
Campus Scientifique de St Jérôme
avenue Escadrille Normandie Niemen
13397 MARSEILLE Cedex 20

Abstract

It is well known that nogood (or lemma) learning can improve combinatorial search. Not all solvers can easily be adapted to learn from failures, as for instance those using complex propagation based constraint programming approaches. This paper reports early experimental results showing that some benefits of learning can be simulated using concurrency, which offers an opportunity of improving solvers having no lookback/learning capacity. We present here a multithreaded framework for the cooperative execution of pure lookahead depth first search algorithms called LRCS (Limited Redundancy Cooperative Search). Experimental results for the SAT problem were obtained using an adapted instance of the Satz Boolean solver. Each cooperative instance of the solver runs in a separate thread in the same process, and potentially explores the entire search tree. Redundancy is limited using a specific state constraint having linear amortized cost. If needed, new threads are dynamically launched based on information gathered by others during the search. This approach shows that the cost of cooperative search is compensated by the associated search savings, which offers a obvious potential for linear speedup on shared memory parallel machines on one hand, and the possibility to improve existing algorithms on the other by the discovery of new heuristics. Each thread is in charge of the whole search space and can thus be stopped at any time. No workload balancing is needed by a true parallel implementation: the first thread to complete stops them all.

Key words : Search, SAT, concurrency, redundancy

1 Introduction

1.1 Cooperative Vs. Concurrent search

We describe here concepts applying to non deterministic depth first search proof procedures like the DPLL procedure [9] for the SAT problem. Each pair (*problem, proof procedure*) defines a search space that can be partitioned into several subspaces. We understand *concurrent* solving as the process of concurrently running several solvers to explore *pairwise disjoint* subspaces as illustrated in figure 1. The field of concurrent problem solving has attracted considerable research interest over years, the main issues being work balancing (work steeling, work sharing) and fault tolerance [4, 3, 12, 13, 1, 2]. We understand *cooperative* solving as the more general process of concurrently running different solvers on the same problem. Unlike concurrent search which is naturally irredundant, cooperative search leads to the exploration of redundant subspaces, as illustrated in figure 2. The focus of this paper is on cooperative search, a field which to the best of our knowledge remained largely unexplored because of this difficulty. Indeed, to prevent redundancy requires the cooperation between solvers via information exchange.

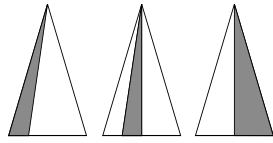


Figure 1: Concurrent Search

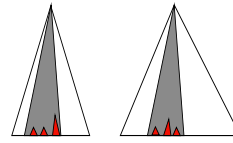


Figure 2: Cooperative Search

1.1.1 Concurrent search issues and benefits

With F a Boolean formula, and a a variable, parallelizing the search simply amounts to solving separately the subproblems $F \wedge a$ and $F \wedge \neg a$. This requires no inter-process communication at all, and of course concurrent machines or clusters may offer in this case a performance improvement linear in the number of machines or processors. This form of parallelization is naturally not fault tolerant (all processes must terminate) and requires workload balancing, since all sub-trees may not have the same size. Because a search tree is seldom balanced (as in figure 3) concurrent search may help discover solutions faster, even when using a single processor as proven in [10, 11], but cannot help however for unsatisfiable problems, unless when lemmas can be exchanged between concurrent solvers.

Different approaches to SAT parallelization have been explored, with or without lemma exchange capabilities. PSATO [3] provides parallelization with workload balancing, PSatz [1] is a concurrent implementation of Satz with dynamic workload balancing by work stealing, PaSAT [2] uses lemma exchange across concurrent solvers. All these tools prevent search redundancy by only using strict parallelism. Of course these approaches benefit a lot from large scale concurrent machines, since only a limited amount of inter-process communication is needed. Efficient workload balancing raise many difficulties of their own in this framework.

1.1.2 Cooperative search issues and benefits

Limiting redundancies in cooperative search requires added communication and algorithmic effort. One obvious way to limit the redundancies to a minimum is to exchange between cooperative solvers the lemmas corresponding to all closed sub-trees (e.g. if the sub-tree corresponding to assignment $a \wedge b \wedge c$ was explored by a solver and no model was found, share the clause $\neg a \vee \neg b \vee \neg c$ across all solvers). To account for a lemma simply requires to add the corresponding clause to the original clause set and continue the search with the new clause set. These lemmas are easily generated. Their size and number are in $\mathcal{O}(V)$, with V the number of variables. Their space complexity is thus in $\mathcal{O}(V^2)$ per solver. They are also very frequently exchanged and locally shortened or suppressed, and their total number (over time) is in $\mathcal{O}(3^V)$ per solver (the size of the set of implicates). These figures are not very favorable even though solvers can postpone lemma sharing until they produced short enough lemma clauses, to limit communication at the extra expense of increasing the number of redundancies. The Limited Redundancy Cooperative Search (LRCS) framework described in the next sections offers extra flexibility in that respect, plus important space and communication savings.

Superlinear improvements using cooperative search have been achieved in some cases using lemma exchange (as for instance for unsatisfiable CSPs in [16]). Figure 4 illustrates the fact that cooperative search can improve search performance by avoiding to repeat many times the exploration of unsatisfiable subspaces. Concurrency did not raise as much research effort as parallelism, because of the added difficulty of dealing with redundancy. LRCS offers a practical solution to redundancy elimination, which renders the implementation of cooperative solvers possible.

That cooperative search offers a potential for improvement over standard single threaded depth first search is obvious if we consider the following extreme case. Let $F = F_1 \wedge F_2$ be a Boolean formula, F_1 and F_2 being built on different vocabularies V_1 and V_2 . If F_1 has many models, F_2 is unsatisfiable, and the DPLL lookahead strategies are fooled enough by the formula structure to complete most of F_1 before doing F_2 ¹ the complexity of the search is in $\mathcal{O}(2^{|V_1|+|V_2|})^2$. Solvers having lemma learning capabilities avoid falling into this trap by detecting that every fail in F_2 is due to F_2 alone. If t concurrent threads are running, the first one rooted in F_1 (this means that the topmost chosen variables are in V_1), and another rooted in F_2 , the complexity falls down to $\mathcal{O}(t * 2^{|V_2|})$. Henceforth, it appears that the potential for a gain in performance using cooperative search at least partially intersects the benefits expected from learning.

¹this situation may dynamically arise during search

²difficult problems often have a large number of big satisfiable sub-formulas

1.2 Motivation and Objectives

This work exclusively deals with thread concurrency, in an attempt to define the simplest possible framework where concurrent and cooperative search can be freely combined, while reducing to a minimum the overhead cost incurred by redundancy elimination in the most flexible manner.

Lookahead strategies for SAT [7] obtain heuristic information from limited searches, that must be interrupted, then undone. This leads to wonder whether these temporary assignments and propagations made for lookahead heuristic needs could be performed by a cooperative solver run freely on the same data.

Some cooperative solvers having lemma exchange capability exhibit superlinear performance speedups with unsatisfiable instances. Producing lemmas requires to maintain run time information about the causes for a failure [14, 17, 6, 5] and to compute the conflicts at every fail. This effort pays back however since from the comparison work in [15], it seems clear that the most efficient SAT solvers are lemma capable (Zchaff [6], Relsat [5]). More convincingly, the limits of pure lookahead depth first backtrack search have been clearly established in [8]. But as our results show, concurrency can be used to further improve ultimately efficient solvers³ not allowing nor using lemma generation like Satz [7] for the SAT problem.

This paper is organized as follows. Section 2 presents the concepts and algorithms allowing to implement the redundancy checks needed in cooperative frameworks. Section 3 describes the LRCS framework, from an algorithmic, heuristic and complexity perspective and an implementation of the redundancy checks in the form of a state constraint. Section 4 presents experimental results and section 5 concludes.

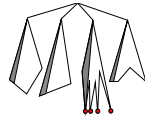


Figure 3: Parallelism Savings

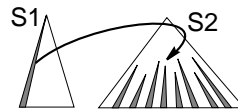


Figure 4: Concurrency Savings

2 Breaking redundancies

A Boolean formula is recursively built from a set V of Boolean *variables*, the logical constants $\{true, false\}$ and the logical *connectives* $\{\vee, \wedge, \neg\}$. An *interpretation* is a mapping from V to $\{true, false\}$. Interpretations naturally extend to mappings from Boolean formulas to expressions in the Boolean algebra. An interpretation I is a *model* (resp. *counter model*) of a formula F iff $I(F) = true$ (resp. $false$). A formula is *valid* (or is a *tautology*) if no interpretation is a counter model. A formula is *inconsistent*, or *unsatisfiable* if no interpretation is a model. Given F a formula, SAT is the problem of proving that F is satisfiable. The notion of a search space is understood with respect to the DPLL [9] enumeration procedure, as listed in figure 5.

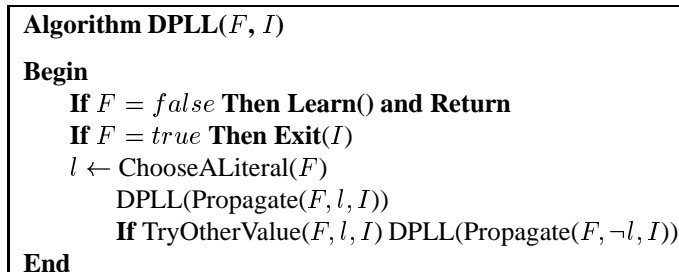


Figure 5: The DPLL algorithm

The parameters F and I to $DPLL$ are a formula and an interpretation. *ChooseALiteral* implements the heuristics. It selects a Boolean variable, and the first value to try, altogether returned as a Boolean literal. *Propagate* is a

³in the sense that variable selection heuristics cannot be expected to improve the programs further, as shown in [8]

two value return function that implements Boolean propagation. *Propagate* returns a simplified formula F' , and an increased partial interpretation $I' \supset I \cup \{l\}$ where every added literal is implied by $F \wedge I$. The result of simplifying F can be *true* or *false*. *TryOtherValue* returns true in general, except when the literal l is pure in F (the negation of l does not occur in F), or information gathered in the left branch allows to backjump over the current node. Note that the return statement in the case F is *false* triggers *backtrack* in the usual sense. Nogoods (or lemmas, or also minimal conflicts) can be obtained by *Learn* every time DPLL fails (i.e. F is *false*).

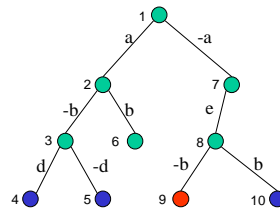


Figure 6: Search Tree

The initial call to DPLL for a given formula F is $DPLL(F, \emptyset)$. An input formula is usually simplified by various polytime preprocessing operations before DPLL is called. According to these definitions, the DPLL call tree maps to a *search tree* where every DPLL call maps to a *node* in the tree and every *edge* is labeled with the propagated literal as in figure 6. Nodes are labeled with the formula and the current interpretation. The set of search tree nodes is traditionally called the *search space*. A path from a given node to its topmost ancestor defines a *search state*.

To prevent redundancies, we use a descriptor of the part of the search space explored by a given solver (later called a *subspace*). This descriptor simply amounts to information naturally present in the solver stack⁴, also called a guiding path in [18].

Definition 1 A search state S_I is a triple (I, R_I, B_I) where I is a partial interpretation, $R_I : I \mapsto \{1, \dots, V\}$ is an injection ranking the elements in I (V is the number of variables in F), and $B_I : I \mapsto \{true, false\}$ is the backtrack status function.

A search state describes a guiding path in the search, augmented with informations about which backtrack points are pending. Technically, a search state simply amounts to a vector of literals (the interpretation *stack* that all DPLL implementations need) and a vector of Booleans (the backtrack status stack). Given a search state S_I and set of literals K , it is very easy to determine whether all/some/none of the interpretations $J \supset K$ are before K . In the illustrations 7 and 8, a left branch means the choice point is not backtracked (the b appended to the variable names has the same meaning).

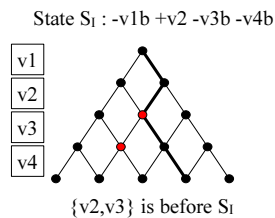


Figure 7: before

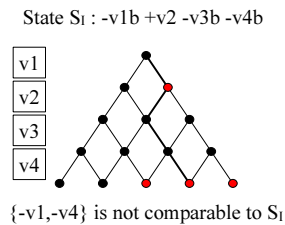


Figure 8: not comparable

Proposition 1 Given S a solver, let S_I be a search state, and K any partial interpretation. That $K \prec_S I$ can be decided in linear time, and requires no knowledge on S .

⁴the data structures required to perform backtrack search: the interpretation stack, the backtrack status stack and the state restore stack

The proof is straightforward and results in the linear time algorithm listed in figure 9. This algorithm exploits the backtrack status of a literal in the interpretation to determine whether a parameter interpretation K is located before or after the reference state S_i .

<pre> Function Compare(S_I, K) Begin forall $l \in S_I$ if $l \in K$ continue if $\neg l \in K$ and l is backtracked in S_I return Before ($K < I$) if $\neg l \in K$ and l is not backtracked in S_I return After ($K > I$) else return Unknown endfor End </pre>

Figure 9: Algorithm for comparing an interpretation and a search state

Search states are used to denote the search space explored by a given solver. Proposition 1 provides the main tool for organizing cooperative search with limited redundancies. To inform other solvers about the work it has done so far only requires every solver to communicate its current search state at regular checkpoints. Each solver only needs to store this search interval to hold the information pertaining to every other solver. Of course, a search state is a compact representation of a list of top level lemmas.

Search states are exchanged across solvers, or stored in shared memory to be exploited later. Both the space and algorithmic cost can be reduced.

Definition 2 Two solvers S_1 and S_2 agree on Boolean propagation iff, for all F a Boolean formula, and $\{l_1, ..l_p\}$ a set of literals, the interpretations I_1 and I_2 built by $Propagate_1(F, \{l_1, ..l_p\}, \emptyset)$ and $Propagate_2(F, \{l_1, ..l_p\}, \emptyset)$ are the same.

- if all solvers agree on Boolean propagation, the literals present in the current interpretation as a result of Boolean propagation need not be listed in a search state. The space needed by a program running N cooperative solvers to hold state information is in $\mathcal{O}(NV)$. This can be reduced for systems having good propagation efficiency since not all variables need to be listed in the search state. This option is not intuitively useful because the more information there is in K , the sooner a backtrack can occur. This is confirmed by experience. In situations where the problem has many variables, and the cost of exchanging the states is too big, this may become a useful tradeoff however.
- the algorithm 9 is a three state machine. It is possible to detect inconsistencies faster by skipping variables missing in the currently compared interpretation, provided their counterpart in the search state is known as backtracked. This possibility is obvious when looking at the figure 7 where the variable v_1 is missing. The resulting state machine is listed in figure 10. Here, plain black arrows represent the case $l \in K$, simple dashed arrows (in the lowest part of the figure) denote the case where l is backtracked in K , and variable dashed arrows (in the topmost part of the figure) represent the case where l is not backtracked. The dotted lines pointing to the state "Unknown" represent all the remaining cases. At start, before the first literal is compared, the automaton state is "Equal". When the state "After" is reached, the constraint becomes frozen until K changes. When the state "Before" is reached, the program backtracks since the currently built interpretation was already explored. When "Unknown" is reached, more information is needed to decide, and the program continues.

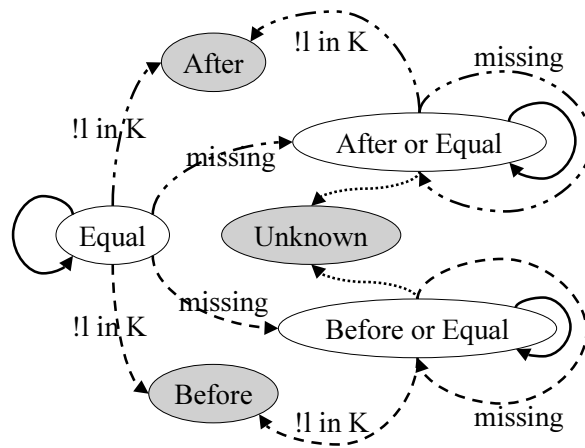


Figure 10: Comparison Automaton

```

Procedure LRCS( $F$ )
Begin
  Start first solver
  When (thread event)
    If model found or done Exit
    Publish new state information
    If thread limit not reached and needed
      Start cooperative solver
  End When
End
    
```

Figure 11: LRCS monitor thread algorithm

3 Limited Redundancy Cooperative Search

The Limited Redundancy Cooperative Search (LRCS) algorithm involves a master monitoring thread holding all the information needed to control the search and to communicate between solvers. Its algorithm, as sketched in figure 11, starts by running a first solver instance. If the search progress is fast enough, no other thread will be launched. When a model is found, or a thread terminates, the monitor stops.

The monitor may create new cooperative threads dynamically. Information naturally maintained by each solver like the number of backtracks can be used for this purpose. Among the probably many useful criteria to discover, we started with a measure of the *difficulty* a solver encounters to close a node. This can be used to start cooperative solvers, which attempt to exploit the existence of potentially strong lemmas.

3.1 Measuring a formula difficulty

We use the following measure of difficulty:

Definition 3 (Difficulty) *Let S be a solver, and F a formula, involving V variables. If the search tree for F involves B backtracks, then the difficulty of F for S is measured as the ratio $V / \log_2(B + 1)$.*

From this definition, since a worst case search tree for V variables has $2^V - 1$ backtracks, the worst case difficulty is 1, and the best case difficulty is V . Of course, the count of yet unassigned variables is known at every node, so that the node difficulty can be computed in $\mathcal{O}(1)$. Note that all solvers normally count the nodes and the backtracks, so that measuring difficulty does not involve complex code modifications.

3.2 Starting cooperative solvers

When a node is closed, if the difficulty of the formula at that point is small, it means that the parent node variable belongs to a lemma approximation. If the difficulty measured improves the previously recorded difficulty by more than some *tolerance*, the monitor starts a cooperative solver exploring the parent path variables in reverse order as is illustrated in figure 12. One argument for reversing is that when a literal assignment triggers an immediate failure by propagation, the only literal known to belong to all conflicts is this one. It is thus realistic to place it at the top of a new cooperative search tree. It also offers a possibility to start new solvers with variables that are not randomly chosen.

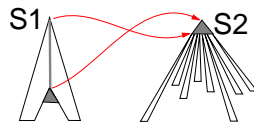


Figure 12: Reversing

3.3 Inter Process communications requirements

Each solver runs at its own pace, and from time to time informs the others about the current state it is in. As seen in the algorithm 9, the state interpretation size has impact on the global performance since the comparison algorithms depend upon this size. Of course, inter-process communication requirements may also suffer from exchanging too often too long state information. A criterion for deciding to publish the current state could hence be the fact that backtracking occurred so that the current partial interpretation is small enough, but we have not tested this issue.

Some interpretations may be explored several times, because of the interval between state publications. Limiting these redundancies requires to reduce the interval between state publications, at the expense of more communications. Experimental results show the impact of this choice on the program statistics.

3.4 State Constraints

The algorithm 9 can be implemented as an incremental propagation based constraint which we call a *state constraint*. Each such constraint incrementally controls the validity of the current interpretation I with respect to a given cooperative solver state S_i . The constraint logic uses an incremental variant of the algorithm 9, where an additional backtrackable cursor serves as an index into the interpretation vector. Each state constraint replaces a corresponding list of lemma clauses that would have been recorded by a solver having learning capabilities. The space needed to store this constraint is linear ($\mathcal{O}(V)$), while it is quadratic for lemma clauses ($\mathcal{O}(V^2)$). Every time a solver publishes a state update, each related constraint must be reset⁵. Figure 13 sketches the algorithm used to account for the insertion of a new literal in the current interpretation, in the simplified three state implementation of the comparison. In this algorithm, the search state S_I is global to several activations of the constraint, as well as the cursor *pos* and the Boolean parameter *active*. K represents the current interpretation of the calling solver. Note that the constraint may be woken only once after several changes in K . This is useful when Boolean propagation fires and pushes several new literals in K , since this process is usually optimized and may not be interrupted easily or usefully. After execution, the constraint may trigger a fail, when the current interpretation belongs to the state interval under review by the remote solver S_i .

⁵the eventually non empty common section between two consecutive states can be exploited to prevent a full reset

```

State Constraint( $K$ )
Begin
  if not active return Inactive
  while true
    select  $l \in S_I$  at position  $pos$ 
    if none: active :=false, return Equal
    if  $l \in K$ :  $pos := pos + 1$ , continue
    if  $\neg l \in K$  and  $l$  backtracked
      active :=false, return  $K < I$ 
    if  $\neg l \in K$  and  $l$  not backtracked
      active :=false, return  $K > I$ 
    else return Unknown
  end while
End

```

Figure 13: State Constraint Algorithm

4 Experiments

We ported to Windows, and adapted to multithreading a public version of Satz215⁶ thanks to Chu Min Li. Our choice was guided by the need for a very optimized pure lookahead SAT solver. Of course, LRCS is compatible with lookback algorithms as well, but our goal is to demonstrate the possibility of improving a near optimal program involving the simplest possible data structures. Satz is one of the fastest pure lookahead search programs to date, and is public. We used the Windows version of the Posix pthreads library. The experimentation platform is a Pentium III portable computer at 900 MHz running Windows XP. In this early version of the program, we made the following choices:

- we used no parallelization in the traditional sense: all threads correspond to truly independent solvers, each potentially exploring the whole search space.
- each solver places a different variable at the top of the search tree. Although it is not clear whether this impacts much, it seems reasonable so that solvers do not step on other's toes.
- state publications occur at regular intervals, every once in several backtracks. This is an important parameter to the program, typically a value between 5 and several hundreds.
- even though we used satisfiable formulas to assess the completeness of our program, we focused on a very specific category of difficult unsatisfiable formulas: the dubois formulas, and specially dubois20 because it can be solved in a few seconds. These formulas are very hard random generated formulas, which exhibit little or no structure to solvers having no capacity to detect equivalences. Multithreaded approaches can probably be more beneficial in the case of logical formulas representing information with structure, but the system properties can be better observed in this arbitrarily difficult case.
- the only reason for starting a thread is that the difficulty ratio was improved by some node.
- threads are only created, and never canceled. This is probably an area of improvement, as the results illustrate.

The dubois20 problem is unsatisfiable and involves 60 variables and 160 clauses initially. Also, it is very symmetrical and contains many equivalences. It is thus a potentially bad test case for LRCS, and will allow to demonstrate the overhead cost of our approach. Satz builds a clause set of size 180 at preprocessing. The results in table 1 show the variations in run time and various other statistics induced by threads. The first line lists the statistics obtained for a reference run of the program running only one thread, and performing no useless computations (like

⁶available at <http://www.laria.u-picardie.fr/cli/EnglishPage.html>

Table 1: Sample output

time	threads	interval	branches	backtracks	fails	tolerance
14	1		524287	262144		
8	7	200	336138	168961	1828	1.05
17	8	200	594334	306969	19635	1.05
9	6	200	417939	213429	8936	1.05
9	5	200	407226	206797	6385	1.05
8	6	200	347103	175369	3647	1.05
10	5	200	422127	219388	16661	1.05
16	8	200	516397	280770	45158	1.05
14	6	200	529363	270358	11383	1.05
15	8	200	496738	261422	26167	1.05
12	8	200	424195	215806	7456	1.05
12	6	200	478748	252916	27100	1.05
11	6	200	448991	227093	5217	1.05
16	5	200	529780	271124	12506	1.05
16	6	200	521346	288731	56148	1.05
9	6	200	419109	211683	4275	1.05
14	7	200	479046	246432	13838	1.05
18	7	200	574280	313967	53678	1.05
11	5	200	412231	210223	8241	1.05
16	8	200	515297	277723	40184	1.05
9	8	200	368009	187026	6082	1.05
12,5	6,55	200	461919	240309	18726	1.05

storing the state). The last line lists the average values for the statistics over all samples. For fairness, it must be mentioned that the fully optimized code corresponding to the original Windows port of Satz runs the same problem in only 10 seconds. The difference stems from the fact that data structures are dynamically allocated and accessed in the multithreaded version, and some thread operations require synchronization (there are two threads in the "one thread" program: the main program, and the only solver thread). It can be observed that our program almost always scans fewer branches and does fewer backtracks. A fail in the "fails" column is counted every time a state constraint triggers a backtrack. In those examples, the interval is set to 200, and the tolerance is 1.05 (a new thread is started when the new difficulty ratio improves the former one by more than 5 per cent). Table 1 shows that the randomness induced by multithreading introduces very important variations. It is clear that the cooperative version of Satz offers superlinear speedup over the "unmodified" original program, even though an average 7 threads are started and run concurrently.

If we vary the interval between state publication operations, we can see that it has little impact on the results over a wide range of values. The results in table 2 list average values computed from 20 successive runs of the same option. Of course, when the interval increases, the threads are more likely to explore redundant subspaces. In fact, there is a clear tradeoff between this redundancy, and the fact that higher intervals result in fewer thread synchronizations.

The *tolerance* parameter controls the launch of a new thread in the following way: a tolerance of 1.05 means that a new thread is started only if it improves the difficulty peak by 5 per cent. This limits the total number of threads, to an average of 7 in the previous runs. The general impact of increasing the tolerance is to let the system wait to start new threads that indeed bring something interesting enough. The pattern of results obtained when varying the tolerance in table 3 illustrate this. The potential interest of this research is better understood with the existence of runs that cut the computation times by a factor 2. This effect of the randomization induced by multithreading suggests the existence of potentially interesting heuristics that still remain to be discovered. One issue may reside in the capacity to decide to interrupt a thread that is currently working with little results to replace it by a more promising one. Table 4 lists exceptionally easy runs obtained with different values of the parameters.

Table 2: Impact of changing the interval

time	threads	interval	branches	backtracks	fails	tolerance
14,8	7,2	5	486458	258529	30625	1,05
13,65	7,1	10	466123	246809	27525	1,05
13,05	6,8	25	464881	242264	19680	1,05
11,55	6,9	50	435986	224573	13192	1,05
12,7	6,5	100	466952	243212	19503	1,05
12,5	6,5	200	461919	240309	18726	1,05
12,8	6,7	500	488072	252158	16273	1,05
12,65	5,9	1000	467005	243189	19396	1,05
12,3	6,6	2500	468343	243600	18884	1,05
13,75	7,2	5000	480545	253708	26910	1,05
15,45	7,4	10000	546520	286378	26279	1,05
16,3	7	20000	594627	306710	18835	1,05

Table 3: Impact of changing the tolerance

time	threads	interval	branches	backtracks	fails	tolerance
21,65	20	50	572280	316948	61695	0,99
13,5	9	50	476089	248446	20845	1,01
13,05	7,25	50	465239	244066	22923	1,03
13,15	6,75	50	478948	248497	18078	1,04
11,55	6,5	50	435986	224573	13192	1,05
13,25	6,45	50	462347	240800	19277	1,08
12,1	6,4	50	456499	237761	19050	1,10
12,5	5,45	50	462565	241496	20447	1,15
12,35	5,65	50	455997	237193	18407	1,20
17,6	5	50	557168	298459	39771	1,30
17,125	4,625	50	538198	290572	42967	1,50

Table 4: Some very positive results

time in millis	threads	interval	branches	backtracks	fails	tolerance
6729	8	50	322129	162854	3615	1.03
6689	6	50	321077	161933	2818	1.05
7102	6	1000	347035	174060	1113	1.05
6874	7	50	253636	128305	3004	1.05
6519	5	50	327435	164805	2188	1.20
6489	6	50	319486	161019	2572	1.20
5537	5	250	294559	147670	795	1.15
5397	5	250	290988	145856	740	1.15
6539	5	250	323676	163184	2705	1.15

Our program has been tested to solve satisfiable formulas, which to some extent assesses its completeness. It is difficult to practically validate concurrent implementations, and we are working on this. The results in table 5 show that problems with many variables like the parity problems are not solved faster in our approach. The current implementation is however not optimal in many aspects, also no parallelism is used, although this is clearly a very important feature to attack satisfiable formulas. The results in table 5 were obtained for a tolerance of 1.05. The top results are obtained using the single threaded option, and the bottom ones are obtained using the multithreaded version. It is clear that extra threads cost too much in this situation, which requires improvement. It should be noted however that very easy instances are solved very fast too in our framework since threads are started dynamically.

Table 5: The parity 8 and parity 16 satisfiable instances

name	time in millis	threads	interval	branches	backtracks	fails	tolerance
par8-1-c.cnf	20	1	50	3	1	0	1.05
par8-2-c.cnf	20	1	50	1	0	0	1.05
par8-3-c.cnf	30	1	50	4	1	0	1.05
par8-4-c.cnf	20	1	50	1	0	0	1.05
par8-5-c.cnf	20	1	50	3	0	0	1.05
par16-1-c.cnf	1351	1	50	1591	794	0	1.05
par16-2-c.cnf	520	1	50	499	245	0	1.05
par16-3-c.cnf	1632	1	50	1489	742	0	1.05
par16-4-c.cnf	350	1	50	415	205	0	1.05
par16-5-c.cnf	10	1	50	1296	644	0	1.05
par8-1-c.cnf	30	1	50	3	1	0	1.05
par8-2-c.cnf	20	1	50	1	0	0	1.05
par8-3-c.cnf	20	2	50	4	1	0	1.05
par8-4-c.cnf	20	1	50	1	0	0	1.05
par8-5-c.cnf	20	1	50	3	0	0	1.05
par16-1-c.cnf	3565	6	50	4588	2400	230	1.05
par16-2-c.cnf	771	3	50	774	384	6	1.05
par16-3-c.cnf	2353	2	50	2162	1112	71	1.05
par16-4-c.cnf	250	4	50	296	145	4	1.05
par16-5-c.cnf	2123	5	50	2718	1397	110	1.05

5 Conclusion

LRCS is a framework for the integration of cooperative solvers where the monitor program can start concurrent or cooperative solvers dynamically. Redundancies are broken using linear amortized cost state constraints, which replace the lemmas produced at the top of search trees with a more compact representation, resulting in more lightweight inter-process communications, space and run time overhead. The amount of unavoidable redundancies can be limited at will by increasing the number of state exchanges. These experiments suggest the existence of new families of heuristics for concurrency based solvers, and prove that in a specially non favorable situation, the benefits of concurrency outweigh the cost of thread synchronization, scheduling and communications. Some known heuristic or simulated parallelism approaches may fall within this scope. LRCS can be adapted to various existing solvers, because the modifications required are limited, since they do not require any change in the internal data structures or algorithms. The code for handling the state constraint need just be inserted in a few places to trigger additional backtracks. Most the information needed, like node counts or variable backtrack status, is readily available. Furthermore, LRCS can be naturally extended to the CSP paradigm, at the higher cost of knowing which elements in the domain of every variable has already been tried. When min domain/max domain heuristics are

used (pick the smallest/highest possible value), storing state information is as easy as for propositional logic. The essential result is that a simple and robust propagation based solver having no lookback capacity, and probably optimal in its performances, can be improved by encapsulating it in a shared memory framework.

Bibliography

1. C. M. Li B. Jurkowiak and G. Utard. Parallelizing Satz using dynamic workload balancing. In *Proceedings of the LICS 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT'2001)*, pages 205–211. Electronic Notes in Discrete Mathematics: <http://www.elsevier.nl/locate/endm/volume9.free>, Jun 2001.
2. W. Blochinger C. Sinz and W. Küchlin. PaSAT : Parallel SAT checking with lemma exchange: Implementation and applications. In H. Kautz and B. Selman, editors, *LICS 2001 Workshop on Theory and Applications of Satisfiability Testing*, volume 9 of *Electronic Notes in Discrete Mathematics*. Elsevier Science Publishers, Jun 2001.
3. M. P. Bonacina H. Zhang and J. Hsiang. PSATO: A distributed propositional prover and its application to quasigroup problems. In *Journal of Symbolic Computation*, volume 21, pages 543–560, 1996.
4. T. Hogg and C. P. Williams. Expected gains from parallelizing constraint solving for hard problems. In *Proc. of 12th Natl. Conf. on Artificial Intelligence (AAAI94)*, pages 1310–1315, Menlo Park, CA, August 1994. AAAI Press.
5. R. J. Bayardo Jr. and R. Schrag. Using CSP look-back techniques to solve exceptionally hard SAT instances. In *Proc. of the Second Int'l Conf. on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science 1118, pages 46–60. Springer-Verlag, 1996.
6. M. Moskewicz L. Zhang, C. Madigan and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the ICCAD 2001 conference, San Jose, CA*, Nov 2001.
7. C. M. LI and Anbulagan. Look-ahead versus look-back for satisfiability problems. In *Proceedings of third international conference on Principles and Practice of Constraint Programming–CP97, Autriche*, Lecture Notes in Computer Science 1330, pages 342–356. Springer-Verlag, 1997.
8. C. M. LI and S. Gerard. On the limit of branching rules for hard random unsatisfiable 3-sat. In *Proceedings of European Conference on Artificial Intelligence (ECAI-2000)*, pages 98–102.
9. G. Logemann M. Davis and D. Loveland. A machine program for theorem proving. In *Communications of the ACM*, volume 5(7), pages 394–397, 1962.
10. P. Meseguer. Interleaved depth-first search. In *Proceedings of the IJCAI'97 Conference*, pages 1382–1387, 1997.
11. P. Meseguer and T. Walsh. Interleaved and Discrepancy Based Search. In *Proceedings of the ECAI'98 Conference*, pages 239–243, 1998.
12. L. Perron. Search procedures and parallelism in constraint programming. In Joxan Jaffar, editor, *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming*, volume 1713, pages 346–360, Alexandria, VA, USA, October 1999.
13. C. Schulte. Parallel search made simple. In Nicolas Beldiceanu, Warwick Harvey, Martin Henz, François Laburthe, Eric Monfroy, Tobias Müller, Laurent Perron, and Christian Schulte, editors, *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, number TRA9/00, pages 41–57, 55 Science Drive 2, Singapore 117599, September 2000.
14. J. P. M. Silva and K. A. Sakallah. Conflict analysis in search algorithms for propositional satisfiability. In *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence*, Nov 1996.
15. L. Simon. <http://www.lri.fr/~simon/satex/satex.php3>. In *Satex Web Site*, 2002.
16. C. Terrioux. Cooperative search and nogood recording. In *Proceedings of the International Joint Conference on Artificial Intelligence IJCAI'01*, 2001.
17. H. Zhang. Lemma generation in the Davis Putnam method. In *Proceedings of the CADE17 Workshop on Model Computation*, Jun 2000.
18. H. Zhang. A random jump strategy for combinatorial search. In *Proc. of International Symposium on AI and Math, Fort Lauderdale, Florida, 2002*, 2002.