Fast Parallel PDR Algorithms for Planning

Marshall Clifton, Charles Gretton
The Australian National University, Canberra, ACT 2601, Australia
{marshall.clifton, charles.gretton}@anu.edu.au

Abstract

SAT lies at the core of *Property Directed Reachability* (PDR), a search paradigm for solving a range of transition system problems without having to reason about huge unwieldy formulae. We consider the application of PDR to the AI planning problem. We develop and extensively evaluate two new strategies for using incremental SAT in parallel computing environments to accelerate classical planning via PDR. Starting from the planning problem's unique initial state, here PDR proceeds by using a SAT solver to iteratively evaluate whether or not a problem state can be progressed towards the goal. If a state can be progressed, then the successor is registered for future evaluation. Otherwise, the reason for the progression failing is inferred using SAT during the evaluation, and that derived reason then constrains future evaluations. Our first parallel strategy, called PS-PDR, employs a single centrally managed priority queue of pending state evaluations, and performs state evaluations in parallel using a pool of incremental SAT solvers. Our second parallel strategy improves the scalability and efficiency of PDR-based planning by exploiting the compositional structure exhibited in some problems. That approach, called PD-PDR, solves subproblems using independent PDR processes, and when available, subplans are linked to form a concrete plan. We experimentally demonstrate that both parallel approaches can achieve substantial runtime gains in a wide range of SAT and UNSAT planning benchmarks.

Introduction

We study the *classical planning* problem. This is the problem of determining whether a finite discrete deterministic transition system can, via a sequence of actions, transition from an initial state to a state satisfying a goal condition. The system is represented succinctly in a formalism such as STRIPS (Fikes and Nilsson 1971) or PDDL (McDermott et al. 1998). A range of solution procedures for this problem have been proposed which are based on the Boolean satisfiability problem (SAT) (Kautz and Selman 1992, 1996; Biere et al. 1999). They proceed by posting a series of SAT queries, each corresponding to a horizon limited version of the planning problem. Recent developments have improved runtime performance using specific tailoring of decision procedures (Rintanen 2012), careful allocation of (parallel) computing resources to queries (Rintanen 2004; Streeter and Smith 2007), and by adopting novel innovations (e.g., incrementality) in SAT solving (Gocht and Balyo 2017). In practice these approaches face two core difficulties: *(i)* it's difficult to prove that no plan exists,[1] and *(ii)* if the smallest plan is very large they can require a lot of memory, because the solution procedure must represent an unrolling of the transition system model over a large number of steps.

PDR is an approach devised to address these difficulties (Bradley 2011; Eén, Mishchenko, and Brayton 2011). It was originally devised for model checking safety properties in hardware verification and has since been extended to a range of other related settings, including planning (Suda 2014). PDR operates by iteratively refining reachability information—a formula describing an overapproximation of the set of states that are N steps away from the goal. This knowledge lies at the heart of PDR, and is used to determine states which may be able to reach the goal. These states are iteratively progressed towards the goal, guided by the reachability information. Either a successor state is found which is closer to the goal, or a generalization of the state is used as a nogood to refine the reachability information. If a sequence of states from the initial state to a state satisfying the goal condition is found, a plan can be extracted from the corresponding transitions. Additionally, if no plan exists, the reachability knowledge can be used to determine this. PDR has been recognised as being highly amenable to parallel computing, there exists portfolio variations for model checking of hardware and software systems (Chaki and Karimi 2016; Marescotti et al. 2017).

In this paper we develop and evaluate two new SAT-based parallel PDR algorithms for classical planning. The first, called PS-PDR, recognises that in planning problems PDR accumulates a very large backlog of states to be progressed, with each progression achieved via a separate query to a SAT solver. Using a centrally managed pool of incremental SAT solvers, we develop an approach that progresses states and, where applicable, derives failure reasons in parallel. SAT queries are conceptualised as units of work, and a central orchestrator schedules that work to be completed in a way that is greedy w.r.t. the planning goal. Our second algorithm,

[1]Given a (tight) completeness threshold the classical *bounded model checking* approach is complete (Jason Baumgartner and Abraham; Kroening et al. 2011; Abdulaziz and Berger 2021).

called PD-PDR, proceeds in a similar manner to portfolio approaches from model checking, though employs a problem compositionality framework from the planning literature that is native to our setting, and thus departs from existing PDR portfolios considerably in terms of the composition of the portfolio. Using dependency analysis, we decompose the concrete problem into a series of subproblems so that a concrete plan corresponds to the concatenation of subproblem plans, and so that each subproblem can be solved independently in parallel. PD-PDR is not a complete procedure, though in planning benchmarks that are susceptible to compositional analysis it is extremely successful. We performed a detailed empirical analysis on both satisfiable and unsatisfiable planning benchmarks, with our analysis focusing on *interesting* problems, which take at least one solver ten seconds to solve. Comparing to a "virtual best" serial baseline that includes PDRPLAN (Suda 2014), using 48 cores our implementation of PD-PDR has a maximal speedup factor exceeding 12.7 in instance *97-1* from the *Logistics* benchmark problem set ($\sim$ 141 seconds v.s. $>$ 30 minutes). It exhibits reliable runtime performance increases across all evaluated domains which feature a compositional structure. Baselining as above, PS-PDR using 48 cores has a maximum speedup factor exceeding 984 ($\sim$ 1.8 seconds v.s. $>$ 30 minutes), in *Bottleneck* benchmark problem number 21.

Background and Notation

Classical Planning

Assuming familiarity with propositional logic we describe the classical planning problem. A problem is given by a tuple $\langle X, A, I, G\rangle$, where X is a set of Boolean-valued state-characterising propositions, A is a set of *actions*, I is the *initial state*, and G is the *goal condition*. A *full* problem state can be described by a *cube*, a conjunctive clause, and specifically a conjunction containing exactly one literal for each element in X. A full state is a representation of a total interpretation over X – i.e., a truth assignment to X. The concept of a partial state is also useful, which is a cube over a subset of the propositions in X, or intuitively, a representation of a partial assignment over X. The initial state I is a full state and the goal condition G is a partial state. Each action $a \in A$ is specified by a *precondition* cube $\mathit{pre}(a)$ and an effect cube $\mathit{eff}(a)$. We assume these cubes are consistent—i.e., if literal ℓ appears in such a cube, then we cannot have $\neg\ell$ also in that cube. An action a is *applicable* to a state s iff $s \vdash \mathit{pre}(a)$. The state $s' = \mathit{succ}(s, a)$ resulting from executing a at s satisfies: $s' \vdash \mathit{eff}(a)$ and for every proposition $f \in X$ absent from $\mathit{eff}(a)$ we have that for all $\ell_f \in \{f, \neg f\}$ if $s \vdash \ell_f$ then $s' \vdash \ell_f$. An action a *conflicts* with another a' if $\nvdash \mathit{eff}(a) \wedge \mathit{eff}(a')$. Those two actions *interfere* if either $\nvdash \mathit{eff}(a) \wedge \mathit{pre}(a')$ or $\nvdash \mathit{eff}(a') \wedge \mathit{pre}(a)$. We say that s_n is *instantaneously reachable* from s, written $\mathit{reachable}^{\forall}(s, s_n)$, if there exists a sequence $[s_0, a_0.., s_{n-1}, a_{n-1}, s_n]$ satisfying: *(i)* $\forall i \in \{1, .., n\}, s_i = \mathit{succ}(s_{i-1}, a_{n-1})$, and *(ii)* for all pairs of actions a_i and a_j we have that these neither conflict nor interfere. In other words, the set of actions $\{a_0, .., a_{n-1}\}$ can be executed in any order at s_0, and the resulting state is s_n. Moreover, we can imagine executing that set of actions instantaneously in parallel at s_0 to reach s_n, as per the $\forall$-step semantics (Rintanen, Heljanko, and Niemelä 2006). We restrict our attention to $\forall$-step semantics in this paper. We call a state s' a *successor* of s iff $\mathit{reachable}^{\forall}(s, s')$ holds. A $\forall$-step *execution* of length n between states s_0 and s_n is a sequence of states $s_0, s_1, .., s_n$ where successive states s_i and s_{i+1} satisfy $\mathit{reachable}^{\forall}(s_i, s_{i+1})$. A planning problem solution is a $\forall$-step execution from I to a state s satisfying $s \vdash G$. Such a solution is called a *plan*.

We will find it convenient to refer to abstract planning problems and problem objects, defined by *restrictions* to a subset $Y \subseteq X$ of problem propositions. For a set of actions A and a set of propositions $Y \subseteq X$, the restriction $A{\downarrow}Y$ is the subset of actions in A that can be described completely if we restrict ourselves to using the symbols from Y only. Writing $\Sigma(f)$ for the set of all propositions mentioned in a formula f, we have:

$$A{\downarrow}Y = \{a \in A | \Sigma(\mathit{pre}(a)) \cup \Sigma(\mathit{eff}(a)) \subseteq Y\} \quad .$$

In the case of a state s and a set of propositions Y, $s{\downarrow}Y$ is the cube restricted to propositions in Y. For example, $(\neg a \wedge \neg b \wedge c){\downarrow}\{a, c\} = (\neg a \wedge c)$

We here assume familiarity with PDR, a sound and complete approach for solving the planning problem (Bradley 2011; Suda 2014).

Problem Decomposition for Planning

We develop a new portfolio-style of PDR algorithm for planning that first decomposes a concrete problem into a sequence of abstract subproblems. A satisfiable concrete planning problem is solved in relatively little time using PDR searches running independently in parallel. The notation and background required for the elicitation of subproblems is described here. We write $\mathit{eff}^+(a)$ for the cube of propositions that occur positively in *eff(a)*, and $\mathit{eff}^-(a)$ for the negative effects. A proposition x might only appear in one sign in the effects of problem actions, either uniformly positively, or uniformly negatively. Let $X^+ \equiv \{x | a \in A, x \in \mathit{eff}^+(a)\}$ be the set of propositions that occur positively, and $X^- \equiv \{x | a \in A, x \in \mathit{eff}^-(a)\}$ the set that occurs negatively. The set of propositions that appear only with one sign in the action effect descriptions is then $X^{\pm} \equiv X \backslash (X^+ \cap X^-)$. The problem *dependency graph*, sometimes called a *causal graph*, was first described by (Knoblock 1994; Williams and Nayak 1997). Departing only slightly from the original conception, we proceed as follows.

Definition 1 (Dependency Graph)**.** *A dependency graph for a planning problem $\langle X, A, I, G\rangle$ is a directed graph (V, E) with a vertex for each proposition in X, and an edge (v, v') for each pair of vertices where:* (i) *There exists an action a where $v, v' \in \Sigma(\mathrm{eff}(a))$ and $v' \notin X^{\pm}$, or* (ii) *There exists an action a such that $v \in \Sigma(\mathrm{eff}(a))$ and $v' \in \Sigma(\mathrm{pre}(a))$.*

For a graph (V, E) with $v, v' \in V$, we write $v \overset{(V,E)}{\rightsquigarrow} v'$ to signify that there is a path from v to v' in (V, E). We also define the *Problem Specific Dependency Graph* (PSDG), a variant of the Dependency Graph which only includes propositions relevant to achieving the goal.

Definition 2 (Problem Specific Dependency Graph)**.** *Given* *a dependency graph (V, E) for problem $\langle X, A, I, G\rangle$, its problem specific dependency graph is obtained by removing all vertices (and thereby their adjacent edges) that are not in the set $\{v' | \exists v \in \Sigma(G), v \overset{(V,E)}{\rightsquigarrow} v'\}$.*

Definition 3 (Strongly Connected Component (SCC))**.** *An SCC of a directed graph is a subgraph in which there is a directed path from each vertex to every other vertex, and every vertex reachable from a vertex in the subgraph is also in the subgraph – i.e., SCCs are maximal.*

Definition 4 (SCC Graph)**.** *Given a dependency graph or PSDG (V, E), its corresponding SCC Graph is a pair $(\mathbf{P}, \mathbf{E})$ where:* (i) $\mathbf{P}$ *is the set of SCCs of* (V, E)*, and* (ii) $\mathbf{E}$ *contains an edge $(\mathbf{p}, \mathbf{p}')$ iff there is an edge $(v, v') \in E$ such that v appears in $\mathbf{p}$ and v' appears in $\mathbf{p}'$. We write $V(\mathbf{p})$ to denote the subset of V that appear in the subgraph $\mathbf{p}$.*

Parallel State PDR

We now describe our first major contribution, a variant of PDR, Parallel State PDR (PS-PDR). PS-PDR is a sound and complete algorithm for solving the planning problem. PS-PDR is identical to PDR in most ways, with the key difference that instead of serially taking obligations from the queue and evaluating them as PDR does, PS-PDR evaluates multiple obligations from one centrally managed queue independently in parallel.

Both PDR and PS-PDR are directed SAT-based approaches, where each SAT variable directly corresponds to the execution of an action or polarity of a proposition. As such, if the problem has a solution it is directly available from satisfying valuations. The algorithms proceed by maintaining and iteratively refining overapproximations of reachability information, called *layer* information about the system. The information at each layer is represented as a CNF formula. A layer formula $\mathcal{L}_i$ is associated with discrete step i, and this is iteratively refined as PDR/PS-PDR proceeds. Such formulae are developed to maintain the invariant that: for every state s such that $s \not\vdash \mathcal{L}_i$ there is no $\forall$-step execution of length i or less from s to a state satisfying G. Initially, $\mathcal{L}_0 = G$ and all other layers are the vacuously satisfied empty CNF formula containing no clauses – i.e., the loosest possible overapproximation. A "*queue*", Q, is maintained with each element a tuple $\langle s, i\rangle$ known as an *obligation*, where s is a state and i is a layer index. Each obligation $\langle s, i\rangle \in Q$ represents a pending query of whether s can reach the goal in i steps. In case the problem is not trivially solvable Q is set up to initially contain a single obligation $\langle I, 1\rangle$—i.e., representing a query of whether the initial state can reach the goal in one step.

Our exposition of PS-PDR makes use of a function $has\_succ(s, \mathcal{L}_{i-1})$, which returns *True* iff the state s has a successor state s' such that $s' \vdash \mathcal{L}_{i-1}$. When computing $has\_succ(s, \mathcal{L}_{i-1})$ on a partial or full state s, a query is made to a SAT solver on a specific one-step formula. That formula corresponds to the cube s, conjoined with the layer formula $\mathcal{L}_{i-1}$, and further conjoined with a formula describing the $\forall$-step semantics transition system model of the problem – i.e., the available actions, their preconditions, and effects. The formula is satisfiable iff a successor state exists, and the query is solved in our implementations using an incremental systematic SAT-solver. The encoding is direct, so if a solution does exist then a successor state is given by the satisfying valuation to the query formula. In that query, s is represented as a list of assumptions, thus if the formula is unsatisfiable, then the *used assumptions* from the incremental solver correspond to a partial state u such that $\neg has\_succ(u, \mathcal{L}_{i-1})$ and $s \vdash u$.

We define the function $get\_succ(s, \mathcal{L}_{i-1})$ on a partial or full state s and formula $\mathcal{L}_{i-1}$, which returns a tuple $\langle exists, s', u\rangle$. Computing this function first involves calling $has\_succ(s, \mathcal{L}_{i-1})$. If this is *True*, the elements of the return tuple are: $exists = \mathit{True}$, $u = NONE$, and s' is a successor of s which satisfies $\mathcal{L}_{i-1}$ extracted from the satisfying assignment from the SAT-solver used to compute $has\_succ(s, \mathcal{L}_{i-1})$. Otherwise, $exists = \mathit{False}$, $s = NONE$, and u is a partial state such that $\neg has\_succ(u, \mathcal{L}_{i-1})$ corresponding to the used assumptions from the SAT-solver. In Alg. 2, we write $\Pi.has\_succ$, $\Pi.get\_succ$ and $\Pi.compute\_reason$ to denote that the transition system used corresponds to problem Π.

Within PDR/PS-PDR when an attempt to find a successor to a state fails—i.e when $has\_succ(s, \mathcal{L}_{i-1})$ is false—a *reason* for failure is derived and added to the layer information at step i. This reason, r, is a partial state such that $s \vdash r$ and $\neg has\_succ(r, \mathcal{L}_{i-1})$. The reason r is added to $\mathcal{L}_i$, by conjoining one disjunctive clause prohibiting states consistent with r, so that for any state y where $y \vdash r$ we have $y \not\vdash \mathcal{L}_i$ for the updated $\mathcal{L}_i$. The state s is itself a reason, but adding $\neg s$ to layer information is not effective. By finding a small reason with fewer literals, the added constraint becomes stronger. In practice, finding reasons with few literals is the most time-consuming operation of both PDR and PS-PDR, but is important for efficiency (Suda 2014).

We formalise reason finding via the function $compute\_reason$, which is invoked in PDR/PS-PDR on an obligation $\langle s, i\rangle$ and the layer information $\mathcal{L}$. It is only defined when $\neg has\_succ(s, \mathcal{L}_{i-1})$. Pseudocode for this is presented in Algorithm 1, in which we use the notation $\ell \in c$ to signify that literal ℓ appears in clause c. Starting with the maximal reason s, the reason is iteratively strengthened by excluding literals not necessary for unsatisfiability. This involves invoking a SAT solver multiple times to evaluate has_succ, each time taking the used assumptions as the revised reason candidate.

With these concepts at hand we now describe PS-PDR. Rather than serially taking obligations from the queue and evaluating them as PDR does, PS-PDR evaluates multiple obligations from one centrally managed queue independently in parallel. This involves running $n > 1$ processes in a distributed computing environment, with one orchestrator and $M = n - 1$ workers. Each worker is given a unique worker ID from $\{1, .., M\}$. All processes keep a local copy of the layer information. Clauses are added to the layer information held by the orchestrator, then disseminated on demand and in increments to workers as they receive work requests. The orchestrator process manages a single queue, a central repository of layer information, and checks for convergence to test if no plan exists. The orchestrator executes

```
Input: A planning problem $\Pi = \langle X, A, I, G\rangle$, an obligation $\langle s, i\rangle$, layer formula $\mathcal{L}$
Output: A reason
1 $r \leftarrow s$
2 for $l \in s$ do
3     if $l \in r$ then
4         $r' \leftarrow \bigwedge_{l' \in r, l' \neq l} l'$
5         $\langle exists, s', u\rangle \leftarrow \Pi.get\_succ(r, \mathcal{L}_{i-1})$
6         if $\neg exists$ then $r \leftarrow u$
7     end
8 end
9 return $r$
```
Algorithm 1: $compute\_reason$

Alg. 3, and each worker executes Alg. 2.

The orchestrator communicates updates about layer information and obligations to the workers, and the workers communicate back: *(i)* reasons for progression failure, or *(ii)* successor states in cases of successful progressions.

The orchestrator and workers communicate via MPI, using the following functions (Clarke, Glendinning, and Hempel 1994; Gabriel et al. 2004). When the orchestrator calls $send\_to\_worker$ with a tuple and a worker ID, that tuple is sent to the corresponding worker, and is retrieved by the worker calling $get\_obligation$. Additionally, $send\_to\_worker$ is documented as taking a layer formula $\mathcal{L}$ as a parameter. This layer information is available to the worker, and stored locally as $\mathcal{L}^{local\_copy}$. The layer information of the workers is only synchronized with the orchestrator when the $get\_obligation$ function is called. The orchestrator does not send the entire layer formulae, but only the incremental difference since the last communication. When a worker calls $send\_[success/failure]\_to\_orchestrator$, the provided tuple is stored in a buffer, which is accessed by the orchestrator calling $get\_[successes/failures]\_from\_workers$, after which, the buffer is emptied. Function $workers\_waiting$ returns the set of worker IDs of processing currently waiting for work having invoked $get\_obligation$. Function $get\_obligation$ is a synchronous call, so the worker blocks until it receives work. Otherwise, $send\_to\_worker$, $send\_[success/failure]\_to\_orchestrator$, $get\_[successes/failures]\_from\_workers$, and $workers\_waiting$ are all asynchronous.

The orchestrator proceeds by first checking if the initial state satisfies the goal condition (Alg. 3, line 2). After this a loop is started over $k \in [1, 2, 3, ...]$ (Alg. 3, line 3). When the orchestrator comes to increment k without having found a plan, $I \nvdash \mathcal{L}_k$ and therefore there is no plan with k or fewer steps. At the start of each iteration, $\mathcal{L}_k$ is the vacuously satisfiable empty CNF formula and $\langle I, k\rangle$ is added to Q (Alg. 3, line 4). The algorithm proceeds by processing obligations from the queue, adding new reachable successors, and adding reasons to layer information.

When elements from the queue are popped off as $\langle s, i\rangle$, elements with minimal i are popped first, breaking ties favouring elements added most recently—i.e, LIFO (Alg. 3, line 8). If the orchestrator receives a tuple $\langle s', s, i\rangle$ from $get\_successes\_from\_workers()$ such that $i - 1 = 0$, then $s' \vdash \mathcal{L}_0$, and therefore s' is a goal state as $s' \vdash G$. As all states mentioned in the queue are either the initial state, or a state that can be reached by the initial state, and s' is a successor to a state which was in the queue, then s' can also be reached from the initial state. Alg. 3, line 14 is able to return *True*, because a goal state reachable from I is found and the planning problem is therefore solved positively.

Whenever a reason r is added to the layer information at $\mathcal{L}_i$, every obligation $\langle s, i\rangle$ in Q such that $s \vdash r$ is removed from Q. Additionally, while not necessary for correctness, when an obligation $\langle s, i\rangle$ such that $i < k$ is sent to a worker to be processed, and the worker returns back that $\neg hassucc(s, \mathcal{L}_{i-1})$, the obligation $\langle s, i+1\rangle$ is added to the queue (Alg. 3, line 18). This is known as *obligation rescheduling* (Suda 2014).

When wrapping up an iteration at k PS-PDR can optionally perform *clause pushing*. In this part of the algorithm pseudocode—lines 21-35 Algorithm 3—we write $c \in f$ to signify that clause c appears in CNF formula f. This is an important processing element for solving planning problems efficiently, and is performed by all the systems we evaluated. This involves considering, for each $i \in \{1, .., k+1\}$, each reason in $\mathcal{L}_{i-1}$ which is not in $\mathcal{L}_i$. Then using it to constrain $\mathcal{L}_i$, if it is a valid constraint.

For each step i in $\{1, .., k+1\}$, there is a check to see if $\mathcal{L}_{i-1} \equiv \mathcal{L}_i$ (Alg. 3, line 34). If the equivalence holds then the goal will never be reachable from the initial state, so no plan exists, so the algorithm returns *False*. Note here that in practice, the check for logical equivalence is a syntactic check—i.e. the check that every clause in $\mathcal{L}_{i-1}$ is in $\mathcal{L}_i$ and vice versa.

```
Input: A planning problem $\Pi = \langle X, A, I, G\rangle$
1 Loop
2     $\langle clause\_pushing, \mathcal{L}^{local\_copy}, \langle s, i\rangle\rangle \leftarrow get\_obligation()$
3     $\langle exists, s', u\rangle \leftarrow \Pi.get\_succ(s, \mathcal{L}^{local\_copy}_{i-1})$
4     if $exists$ then
5         if $\neg clause\_pushing$ then $send\_success\_to\_orchestrator(\langle s', s, i\rangle)$
6     else
7         if $clause\_pushing$ then
8             $send\_failure\_to\_orchestrator(\langle NONE, s, i\rangle)$
9         else
10            $r \leftarrow \Pi.compute\_reason(\langle u, i\rangle, \mathcal{L}^{local\_copy})$
11            $send\_failure\_to\_orchestrator(\langle r, s, i\rangle)$
12        end
13    end
14 EndLoop
```
Algorithm 2: PS-PDR Worker

```
1  $Q \leftarrow \{\}, \mathcal{L}_0 \leftarrow G$, for $j > 0 : \mathcal{L}_j \leftarrow True$
2  if $I \vdash \mathcal{L}_0$ then return True
3  for $k \in [1, 2, 3, ...]$ do
4      $Q \leftarrow \{\langle I, k\rangle\}$
5      while $Q \neq \{\}$, or
        $workers\_waiting() \neq \{1, .., M\}$ do
6          foreach $w \in workers\_waiting()$ do
7              if $Q \neq \{\}$ then
8                  $\langle s, i\rangle \leftarrow$ pop most recently added
                    obligation from $Q$ with minimal $i$.
9                  $send\_to\_worker(\langle False, \mathcal{L}, \langle s, i\rangle\rangle, w)$
10             end
11         end
12         foreach $\langle s', s, i\rangle \in$
            $get\_successes\_from\_workers()$ do
13             $Q \leftarrow Q \cup \{\langle s, i\rangle, \langle s', i - 1\rangle\}$
14             if $i - 1 = 0$ then return True
15         end
16         foreach
            $\langle r, s, i\rangle \in get\_failures\_from\_workers()$
            do
17             for $j \in \{0, .., i\}$ do $\mathcal{L}_j \leftarrow \mathcal{L}_j \wedge \neg r$
18             if $i < k$ then $Q \leftarrow Q \cup \{\langle s, i + 1\rangle\}$ /*
                Obligation Rescheduling */
19         end
20     end
21     for $i \in \{1, .., k + 1\}$ do /* Clause Pushing */
22         CP $\leftarrow \{\langle \neg c, i\rangle \,|c \in \mathcal{L}_{i-1}, c \notin \mathcal{L}_i\}$
23         while $CP \neq \{\}$ or
            $workers\_waiting() \neq \{1, .., M\}$ do
24             foreach $w \in workers\_waiting()$ do
25                 if $CP \neq \{\}$ then
26                     $\langle \neg c, i\rangle \leftarrow$ pop obligation from
                        $CP$
27                     $send\_to\_worker(\langle True, \mathcal{L}, \langle \neg c, i\rangle\rangle, w)$
28                 end
29             end
30             foreach $\langle \_, \neg c, \_\rangle \in$
                $get\_failures\_from\_workers()$ do
31                 $\mathcal{L}_i \leftarrow \mathcal{L}_i \wedge c$
32             end
33         end
34         if $\mathcal{L}_{i-1} \equiv \mathcal{L}_i$ then return False
35     end
36 end
```

Algorithm 3: PS-PDR Orchestrator

Parallel Decomposition PDR

Our second major contribution is the Parallel Decompositional PDR (PD-PDR) algorithm, a sound but not complete parallel PDR algorithm for the planning problem. It involves performing dependency analysis on the problem, to find and represent its different subproblems. These subproblems are then solved for, independently in parallel, afterwards, the results are combined by concatenating subproblem plans to form a concrete plan. The constraints associated with subproblems are determined prior to any plan search. Thus, instead of planning for each subproblem serially, one after the other, they can all be planned for independently in parallel.

The algorithm involves creating a list of planning problems from the original problem. We will refer to list elements as *subproblems*, and the original planning problem as the *concrete problem*. This is done by creating various subsets of propositions, and restricting concrete problem elements to them.

Intuitively, these subproblems are generated so that the initial state of one subproblem roughly corresponds to the goal condition of the previous subproblem, with the first subproblem's initial state being that of the concrete problem, and the goal condition of the final subproblem being that of the concrete problem. Each subproblem considers a subset of the propositions, then attempts to achieve part of the concrete goal condition. Additionally, the subproblems are constrained so that the propositions from the initial state mentioned in later subproblems must be returned to their original polarity in the goal; so that subproblem plans can be concatenated. For propositions that can only change polarity once, it is impossible to achieve such a subproblem goal if a sequenced action changes the polarity. In this case, the above constraint is not imposed, allowing such a proposition to have a new parity in the subproblem goal, we call such propositions *exclusions*. Potentially, multiple subproblem plans could change the same proposition, in which case the concatenation of subplans may not produce a valid concrete plan. PD-PDR is sound if a potential plan is found, it is verified before PD-PDR declares a plan exists. PD-PDR is not complete, as goals from multiple subproblems may need to be achieved together with a single use resource. For a problem where a plan exists, it is not known beforehand whether PD-PDR will be able to find such a plan. In practice however, we find it quite effective.

For a given concrete problem $\langle X, A, I, G\rangle$, the list of subproblems of interest is obtained as follows:

- Let $\Gamma = (\mathbf{P}, \mathbf{E})$ be the SCC Graph of the PSDG (V, E) of the concrete problem.
- Let Δ_G be the list of SCCs in $\mathbf{P}$ that contain a goal proposition. If there is a path from $\mathbf{p}_1$ to $\mathbf{p}_2$ in Γ then $\mathbf{p}_2$ must appear after $\mathbf{p}_1$ in Δ_G.[2]
- Let $\mathbf{p}_G$ be an element in Δ_G, then:

$$F(\mathbf{p}_G) \equiv \{f|\mathbf{p}' \in \mathbf{P}, \mathbf{p}_G \overset{\Gamma}{\rightsquigarrow} \mathbf{p}', f \in V(\mathbf{p}')\} \cup V(\mathbf{p}_G)$$

Intuitively, $F(\mathbf{p}_G)$ contains most of the concrete problem propositions that are required to plan for the goals in $\mathbf{p}_G$. To get the propositions "missing" from $F(\mathbf{p}_G)$ to plan for the goal in $\mathbf{p}_G$, we require the following *exclusions*:

$$\begin{aligned} Ex(\mathbf{p}_G) \equiv \quad & \{x|x \in X^{\pm}, v \in F(\mathbf{p}_G), a \in A, \\ & x \in \Sigma(\mathit{eff}(a)), v \in \Sigma(\mathit{eff}(a))\} \end{aligned}$$

Intuitively, these exclusions are additional propositions—perhaps useful for achieving a subgoal—that can change their polarity at most once.

[2]Note, when the ordering is underconstrained and multiple orderings are possible, in practice we break ties using a lexicographic ordering over subproblems.

- Let $idx(\mathbf{p})$ be the index in Δ_G at which the subproblem $\mathbf{p}$ occurs. We then define $\Phi(\mathbf{p})$ as:

  $$\Phi(\mathbf{p}) \equiv \{f|\mathbf{p}' \in \Delta_G, j = idx(\mathbf{p}'), j > idx(\mathbf{p}), f \in F(\mathbf{p}')\}$$

  Intuitively, $\Phi(\mathbf{p})$ is the union of $F(\mathbf{p}')$ for each $\mathbf{p}' \in \Delta_G$ that occurs after $\mathbf{p}$ in Δ_G.
- A list of subproblems is created. For each $\mathbf{p} \in \Delta_G$, where $X(\mathbf{p}_G) = F(\mathbf{p}_G) \cup Ex(\mathbf{p}_G)$, there is a corresponding subproblem:

  $$\langle X(\mathbf{p}_G), A{\downarrow}X(\mathbf{p}_G), I{\downarrow}X(\mathbf{p}_G), G{\downarrow}V(\mathbf{p}_G) \wedge I{\downarrow}\Phi(\mathbf{p})\rangle$$

Each of the subproblems in this list is then solved independently in parallel with serial PDR. If any subproblem does not have a solution, then the algorithm returns *unknown*. If a plan is found for each subproblem, these plans are concatenated together in the same order as their corresponding subproblems appear in the subproblem list. This potential plan is verified. If it is a valid plan, the algorithm returns *True*, otherwise it returns *Unknown*.

Experiments

We implemented PS-PDR and PD-PDR building from the PDDL parser from the SAT-based planning system MADAGASCAR (Rintanen 2012).[3] We benefit from the preprocessing of that tool, including problem grounding, invariants—i.e., mutex relationships between state propositions—and detection of unsatisfiability in case that is trivial. We implemented a comparison serial PDR solver we call PDR-S. We also baseline our implementations by comparing them with PDRPLAN. LINGELING (Biere 2010) is used as the incremental SAT solver in all our implementations. In the case of PDR-S, the SAT solvers are maintained on a single process, whereas in PS-PDR, workers each maintain their own set of SAT solvers and communicate with the orchestrator using MPI (Clarke, Glendinning, and Hempel 1994; Gabriel et al. 2004). We performed preliminary experiments evaluating our various PDR algorithms with and without obligation rescheduling. We found obligation rescheduling increased the efficiency of the parallel and non-parallel algorithms, aligning with the findings of (Suda 2014). Because of this, all solvers evaluated here use obligation rescheduling. In the case of PD-PDR we first perform the described decomposition and create a list of subproblems. Each of these subproblems is then solved independently in parallel by PDR-S. If planning is successful in all subproblems, these plans are concatenated and the resulting concrete plan has its validity tested using VAL (Howey, Long, and Fox 2004).

Existing parallel PDR approaches in different contexts involve a portfolio of PDR processes each managing their own queue, periodically posting learnt reasons to each other (Bradley 2011; Marescotti et al. 2017; Chaki and Karimi 2016). Each of these existing approaches do not use obligation rescheduling. As these existing approaches cannot parse benchmark planning problems, and to test this portfolio approach with obligation rescheduling, the portfolio approach was reimplemented. The implementation is a variation of PS-PDR, where instead of the orchestrator managing a single queue, a set of queues is managed, one for each worker. Then when a worker requests/produces a state, it is retrieved/added to its corresponding queue. Obligation rescheduling is enabled. We refer to this implementation as PDR Portfolio (PDR-P).

Each PS-PDR/PDR-P run uses 48 cores with one thread per core, so there are 47 workers and one orchestrator. We evaluate the performance of solvers on International Planning Competition (IPC) problems. We also evaluate on instances of the OPTICAL-TELEGRAPHS and PHILOSOPHERS domains created by the PDDL generator from (Fabre et al. 2010). In addition to standard "satisfiable" benchmarks, we include "unsatisfiable" problems from the 2016 IPC Unsolvability track.

All evaluations were performed on an Intel(R) Xeon(R) Gold 6252 CPU with 187GBs Memory. Additionally, as many instances from the unsolvability track timed out, if 5 successive instances of a domain timed out for a particular solver, we report that all successive instances also timed out. We label a planning domain *decomposable* if it is susceptible to the decompositions used in PD-PDR. As PD-PDR is unable to prove a problem has no solution, we do not perform PD-PDR on problems from the unsolvability track.

All problem instances were run with a 30-minute (1800s) timeout. We exclude reporting on the domains: BAG-BARMAN, BAG-GRIPPER, OVER-TPP and SLIDING-TILES from the unsolvability track as none of the variants of PDR we evaluated were able to produce UNSAT proofs in any instances from these—i.e all solvers timed out, were unable to parse the problem or ran out of memory.

Table 1 shows the number of instances solved for each domain (coverage), along with a *speedup factor*, shown as: $\text{instances\_solved}/\text{speedup\_factor}_{\#\text{ratios\_averaged}}$. For results of parallel solvers, the speedup factor is the geometric mean of the runtime ratio between the fastest serial planner (out of both PDRPLAN and PDR-S) and the parallel solver. In the reporting of this factor we only consider problems where: *(i)* both solvers completed successfully, and *(ii)* at least one takes longer than ten seconds. #ratios_averaged is the number of ratios averaged to calculate this factor. A speedup factor is also reported for PDR-S, with PDRPLAN being the comparison solver. When there are no problems matching this criteria, in place of a speedup factor "-" is reported. The best coverage for each domain is bolded. For the parallel solvers, the best parallel speedup is bolded,

The first column reports the domain with the total number of problems and number of *interesting* problems, where an instance is deemed interesting if at least one evaluated solver exhibited a runtime of at least ten seconds. Column Abbr. shows the domain abbreviations we have adopted for later use in Figure 1. Column $\overline{\#CPUs}$ reports the maximum of the number of CPUs allocated to a PD-PDR problem. Because the number of processes needed to perform PD-PDR is sometimes greater than the number of cores available on the testing system, the parallel runtime is simulated via a batch of serialised invocations, thus we report the longest subproblem runtime plus parsing and decomposition time. Domains that do not include a result for PD-PDR and

[3]Source code at: https://github.com/ANU-HPC/parallel-pdr

$\overline{\#CPUs}$ were not decomposable.

Figure 1 records (in logscale) statistics of the distributions of experimental runtimes, either by directly reporting runtimes as points, or summarizing the runtime distributions using a box-and-whisker element. We restrict the distributions under consideration to interesting instances. Some of the distribution elements are lower bounds, as we include a timeout datapoint (1800 seconds) in case: *(i)* of a timeout, *(ii)* the system exhausting memory prior to solving the instance, or *(iii)* in the case PD-PDR is unable to find a valid plan. When presented as a box and whisker plot, the top and bottom of each plot shows the maximum and minimum runtimes. The top and bottom of each box shows the 75th and 25th percentile, and the line inside each box shows the median. The plotted elements in Figure 1 order (left-to-right) and colour the data for the solvers as follows. *(i)* □ PDRPLAN *(ii)* □ PDR-S *(iii)* □ PDR-P *(iv)* □ PS-PDR *(v)* □ PD-PDR (only when decomposable) We plot a gray bar in place of a distribution summary if that solver was not able to parse problems from the indicated domain. The timing data for the Unsolvability track instances are plotted directly, as since many instances time out, box and whisker plot would be heavily skewed and uninformative.

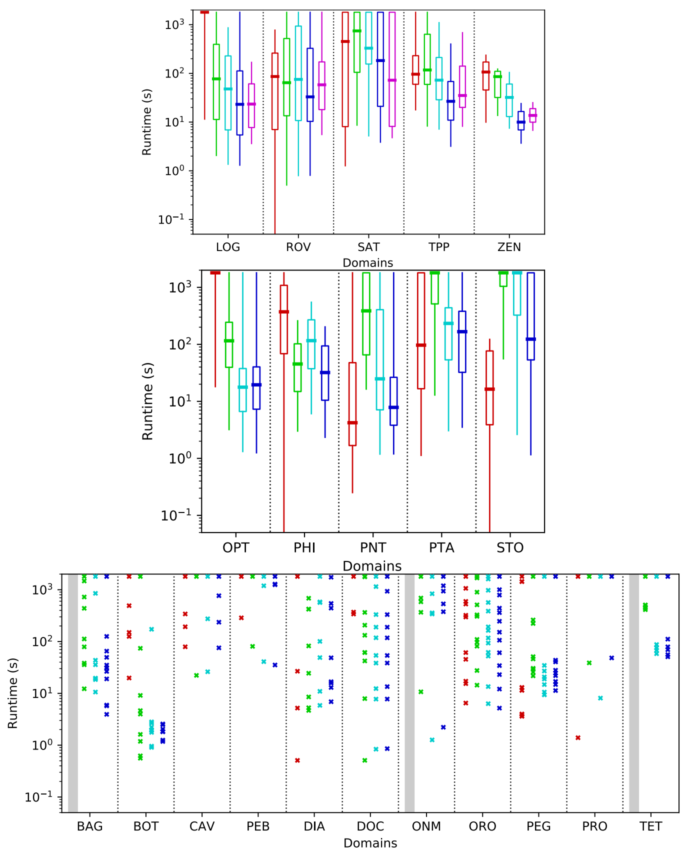

Figure 1: Per-domain runtime distribution summaries. Above each indicated domain, from left to right, the runtimes are for solvers □ PDRPLAN, □ PDR-S, □ PDR-P, □ PS-PDR, □ PD-PDR.

Summarising our evaluation of the parallel solvers: We find PD-PDR (when it is applicable) and PS-PDR to be the fastest solvers. in 49 cases, PS-PDR/PDR-P exhausted the memory of the host, no other solver did this. In only one occasion, PD-PDR was not able to find a plan due to its incompleteness, as opposed to a timeout or memory constraint—subproblem plans were found, but the combination was invalid. This was in the ROVERS domain. For understanding the parallel performance gains of PD-PDR, we note that if we exclude PDRPLAN runtimes, and weigh all domains equally, PD-PDR has a speedup factor of 4.4 over PDR-S. Similarly, PS-PDR has a 2.9 speedup factor over PDR-S for the Unsolvability domains, and 5.7 in the other domains.

Related Work

The first PDR algorithm was IC3, which stood for *"Incremental Construction of Inductive Clauses for Indubitable Correctness"* (Bradley 2011). The algorithm is a SAT-based procedure for reasoning about problem *safety* without explicitly unrolling the transition relation. IC3 featured in the 2010 Hardware Model Checking Competition (HWMCC'10), and Property Directed Reachability was a phrase later coined in (Eén, Mishchenko, and Brayton 2011). A range of serial PDR procedures were subsequently adapted, developed, and evaluated for planning(Suda 2014). Of special interest here is PDRPLAN, a fast bespoke implementation of PDR that takes advantage of the common structure of many classical planning benchmarks—e.g., uniformly positive action preconditions—replacing all SAT inference with specialised constraint processing procedures that perform guaranteed polynomial time inference.

The question of how to use parallel computing to improve the runtime of PDR is of interest in our setting. In the original IC3 manuscript (Bradley 2011), the author, Bradley, recognised the potential for accelerating PDR using parallel computing. He introduced a portfolio scheme, using a small set of IC3 processes that synchronise so that clause pushing (Alg. 3, lines 21-35) can be performed by one serial process, and problem *safety* (Alg. 3, line 34) detected by that process where applicable. In this original work, search diversity is enhanced using a nondeterministic SAT-solver ZCHAFF (Vizel, Weissenbacher, and Malik 2015). IC3 portfolio members share reasons periodically via a central coordinating process and, when an IC3 instance receives reasons found by other processes, it removes all obligations from its queue that are inconsistent with those reasons. With these modifications Bradley was able to demonstrate that a portfolio using 12 cores can complete an additional 12 proofs in HWMCC'10 competition settings, compared with a serial IC3 baseline.

Preliminary investigations of portfolio PDR are extended in (Chaki and Karimi 2016), where a variety of strategies for parallel portfolios with reason sharing between members are described. Designed for hardware model checking problems, which can have many initial states, each portfolio member is a variant of IC3 that uses the deterministic SAT solver MINISAT (Eén and Sörensson 2004). Portfolio members perform their own clause pushing, thus there is no need to synchronise the individual portfolio elements.

Domain (#Interesting/#Total)	Abbr.	PDRplan	PDR-S	PDR-P	PS-PDR	PD-PDR	$\overline{\#CPUs}$
Logistics (158/198)	LOG	63	$197/-$	$\mathbf{198}/1.71_{109}$	$197/2.71_{102}$	$\mathbf{198}/\mathbf{5.08}_{103}$	101
Rovers (16/40)	ROV	40	$\mathbf{39}/0.65_{10}$	$37/0.53_{8}$	$38/1.19_{9}$	$\mathbf{39}/\mathbf{1.68}_{11}$	70
Satellite (20/36)	SAT	**28**	$\mathbf{28}/0.43_{5}$	$\mathbf{28}/1.02_{6}$	$27/\mathbf{3.29}_{5}$	$\mathbf{28}/3.24_{5}$	232
TPP (14/30)	TPP	29	$29/0.8_{11}$	$\mathbf{30}/1.15_{13}$	$\mathbf{30}/\mathbf{3.23}_{10}$	$\mathbf{30}/2.07_{12}$	21
Zenotravel (6/20)	ZEN	**20**	$\mathbf{20}/1.58_{5}$	$\mathbf{20}/1.89_{4}$	$\mathbf{20}/5.55_{3}$	$\mathbf{20}/\mathbf{5.6}_{4}$	26
Optical Telegraphs (153/165)	OPT	38	$\mathbf{153}/21.48_{16}$	$\mathbf{153}/\mathbf{6.55}_{87}$	$\mathbf{153}/5.94_{91}$		
Philosophers (13/15)	PHI	11	$\mathbf{15}/16.97_{6}$	$\mathbf{15}/0.41_{10}$	$\mathbf{15}/\mathbf{1.36}_{10}$		
Pipesworld Notankage (33/50)	PNT	46	$41/0.13_{4}$	$44/0.79_{6}$	$\mathbf{47}/\mathbf{0.91}_{4}$		
Pipesworld Tankage (40/50)	PTA	**39**	$24/0.25_{6}$	$37/0.68_{17}$	$\mathbf{39}/\mathbf{0.98}_{20}$		
Storage (15/30)	STO	30	$21/0.01_{1}$	$21/0.04_{1}$	$\mathbf{25}/\mathbf{0.11}_{4}$		
Bag Transport (27/29)	BAG	0	$10/-$	$8/3.12_{5}$	$\mathbf{13}/\mathbf{9.38}_{5}$		
Bottleneck (9/25)	BOT	20	$24/6.64_{1}$	$\mathbf{25}/\mathbf{0.43}_{1}$	$\mathbf{25}/-$		
Cave Diving (23/25)	CAV	5	$\mathbf{3}/15.33_{1}$	$\mathbf{3}/\mathbf{0.85}_{1}$	$\mathbf{3}/0.29_{1}$		
Chessboard Pebbling (21/23)	PEB	**3**	$\mathbf{3}/3.56_{1}$	$\mathbf{3}/1.97_{1}$	$\mathbf{3}/\mathbf{2.29}_{1}$		
Diagnosis (8/20)	DIA	15	$20/0.32_{1}$	$\mathbf{17}/0.56_{2}$	$\mathbf{17}/\mathbf{52.76}_{1}$		
Document Transfer (18/20)	DOC	4	$11/1.07_{2}$	$\mathbf{10}/\mathbf{1.42}_{6}$	$\mathbf{10}/1.41_{6}$		
Over Nomystery (23/24)	ONM	0	$5/-$	$\mathbf{2}/-$	$\mathbf{2}/-$		
Over Rovers (13/20)	ORO	17	$17/0.62_{9}$	$\mathbf{15}/1.12_{7}$	$14/\mathbf{1.39}_{6}$		
Pegsol (18/24)	PEG	16	$\mathbf{14}/0.14_{4}$	$\mathbf{14}/\mathbf{0.49}_{4}$	$\mathbf{14}/0.4_{4}$		
Pegsol Row5 (12/15)	PRO	4	$\mathbf{3}/-$	$\mathbf{3}/-$	$\mathbf{3}/-$		
Tetris (20/20)	TET	0	$\mathbf{5}/-$	$\mathbf{5}/6.42_{5}$	$\mathbf{5}/\mathbf{6.59}_{5}$		

Table 1: Coverage and average runtime speedup factors.

In the two best strategies described, portfolio members are required to check if the portfolio has proved the problem *safe*. One strategy requires each portfolio element to derive their own proof, and another has portfolio elements consider the cumulative knowledge of all portfolio members, checking if the problem is *safe* when the process is due to increment k. The headline contribution is a detailed statistical and empirical analysis of portfolios in hardware benchmarks. In (Chaki and Karimi 2016), the exploration of portfolios in software verification is left as future work, and that challenge is taken up in (Marescotti et al. 2017). Those authors develop a divide-and-conquer portfolio approach using Spacer (Komuravelli, Gurfinkel, and Chaki 2014), "*partitioning*" the problem at hand syntactically into a set of subproblems, so that the concrete problem is *safe* iff every subproblem is *safe*. In addition to the innovation of partitioning, the authors also develop a "heuristic" for how members of the portfolio share reasons and search.

Our approach to decompositional planning using PDR is based on the dependency graph concept that was first described in (Knoblock 1994; Williams and Nayak 1997). Intuitively, that idea is to synthesise a concrete plan through a process of iteratively refining plans using the abstraction hierarchy associated with the causal graph. These seminal ideas were advanced in concert with literature related to (tree) decompositions (Darwiche 2001; Huang and Darwiche 2003; Robertson and Seymour 1991), with the planning literature developing conceptual frameworks and algorithms that fall under the banner of *factored* planning (Amir and Engelhardt 2003; Brafman and Domshlak 2006). These ideas are showcased and contrasted in relation to the dTreePlan planning system in (Kelareva et al. 2007). Unlike factoring/abstraction-refinement, our approach forms concrete plans by a simple concatenation operation, and not by sub-plan/action interleaving. Our approach is enabled because we "edit" the goals of abstract subproblems, thereby ensuring that—at least in practice on common planning benchmarks—our subplan concatenation operation yields a valid solution to the concrete problem at hand. In this last respect, our contribution is related to (Abdulaziz, Norrish, and Gretton 2015), a work in which the authors employ a goal editing idea to plan in systems composed of a set of symmetric subsystems.

Conclusions and Future Work

We present two new parallel PDR algorithms designed to decrease runtime by using multiple CPUs. Our evaluation of these parallel algorithms shows a significant runtime reduction compared to serial PDR algorithms. We also demonstrate compelling performance gains comparing to a parallel portfolio baseline. Our PS-PDR algorithm exhibits a single centralised queue of obligations, and proceeds by farming the required SAT queries to a fixed size pool of workers, which process the queries independently in parallel. Workers service multiple formulae, one for each PDR layer and are based on incremental SAT solving. Thus, workers benefit from accumulated knowledge gleaned from successive search exercises, and access to used assumptions in reason finding. Our other algorithm, PD-PDR, uses a problem decomposition based on the dependency graph. Other decompositions exist in the planning literature, and we expect these to be worth exploring in the future. We also note that PD-PDR can be used with a different solver taking the place of PDR, including PS-PDR. Combining our parallel algorithms in this way could give further performance increases.

References

Abdulaziz, M.; and Berger, D. 2021. Computing Plan-Length Bounds Using Lengths of Longest Paths. In In Proc. *AAAI*, 11709–11717.

Abdulaziz, M.; Norrish, M.; and Gretton, C. 2015. Exploiting symmetries by planning for a descriptive quotient. In *Proc. of the 24th IJCAI*, 25–31.

Amir, E.; and Engelhardt, B. 2003. Factored planning. In *IJCAI*, volume 3, 929–935. Citeseer.

Biere, A. 2010. Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. *FMV Technical Report 10/1* Institute for Formal Models and Verification, Johannes Kepler University.

Biere, A.; Cimatti, A.; Clarke, E. M.; and Zhu, Y. 1999. Symbolic Model Checking without BDDs. In In Proc.*TACAS*, 193–207.

Bradley, A. R. 2011. SAT-based model checking without unrolling. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, 70–87. Springer.

Brafman, R. I.; and Domshlak, C. 2006. Factored planning: How, when, and when not. In *AAAI*, volume 6, 809–814.

Chaki, S.; and Karimi, D. 2016. Model checking with multi-threaded IC3 portfolios. In In Proc. *VMCAI*, 517–535.

Clarke, L.; Glendinning, I.; and Hempel, R. 1994. The MPI Message Passing Interface Standard. In Decker, K. M.; and Rehmann, R. M., eds., *Programming Environments for Massively Parallel Distributed Systems*, 213–218.

Darwiche, A. 2001. Recursive conditioning. *Artif. Intell.* 126(1-2): 5–41. doi:10.1016/S0004-3702(00)00069-2.

Eén, N.; Mishchenko, A.; and Brayton, R. K. 2011. Efficient implementation of property directed reachability. In In Proc. *FMCAD*, 125–134.

Eén, N.; and Sörensson, N. 2004. An Extensible SAT-solver. In Giunchiglia, E.; and Tacchella, A., eds., *Theory and Applications of Satisfiability Testing*, 502–518.

Fabre, E.; Jezequel, L.; Haslum, P.; and Thiébaux, S. 2010. Cost-optimal factored planning: Promises and pitfalls. In in Proc. *ICAPS*.

Fikes, R. E.; and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence* 2(3-4): 189–208.

Gabriel, E.; Fagg, G. E.; Bosilca, G.; Angskun, T.; Dongarra, J. J.; Squyres, J. M.; Sahay, V.; Kambadur, P.; Barrett, B.; Lumsdaine, A.; Castain, R. H.; Daniel, D. J.; Graham, R. L.; and Woodall, T. S. 2004. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proc. 11th European PVM/MPI Users' Group Meeting*, 97–104.

Gocht, S.; and Balyo, T. 2017. Accelerating SAT Based Planning with Incremental SAT Solving. In In Proc. *ICAPS*, 135–139.

Howey, R.; Long, D.; and Fox, M. 2004. VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL. In *Proceedings - ICTAI*, 294– 301.

Huang, J.; and Darwiche, A. 2003. A structure-based variable ordering heuristic for SAT. In *IJCAI*, volume 3, 1167–1172.

Jason Baumgartner, A. K.; and Abraham, J. A. ???? Property Checking via Structural Analysis. In *Proc. 14th International Conference, Computer Aided Verification, 2002*, Lecture Notes in Computer Science, 151–165.

Kautz, H. A.; and Selman, B. 1992. Planning as Satisfiability. In In Proc.*ECAI*, volume 92, 359–363.

Kautz, H. A.; and Selman, B. 1996. Pushing the Envelope: Planning, Propositional Logic and Stochastic Search. In *In Proc. AAAI*, 1194–1201.

Kelareva, E.; Buffet, O.; Huang, J.; and Thiébaux, S. 2007. Factored Planning Using Decomposition Trees. In *IJCAI*, 1942–1947.

Knoblock, C. A. 1994. Automatically generating abstractions for planning. *Artificial Intelligence* 68(2): 243–302.

Komuravelli, A.; Gurfinkel, A.; and Chaki, S. 2014. SMT-Based Model Checking for Recursive Programs. In Biere, A.; and Bloem, R., eds., *Computer Aided Verification*, 17–34.

Kroening, D.; Ouaknine, J.; Strichman, O.; Wahl, T.; and Worrell, J. 2011. Linear completeness thresholds for bounded model checking. In *Computer Aided Verification*, 557–572.

Marescotti, M.; Gurfinkel, A.; Hyvärinen, A. E. J.; and Sharygina, N. 2017. Designing parallel PDR. In In Proc.*FMCAD*, 156–163.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL: The Planning Domain Definition Language. Technical report, CVC TR-98-003/DCS TR-1165.

Rintanen, J. 2004. Evaluation Strategies for Planning as Satisfiability. In In Proc. *ECAI*, 682–687.

Rintanen, J. 2012. Planning as satisfiability: Heuristics. *Artificial intelligence* 193: 45–86.

Rintanen, J.; Heljanko, K.; and Niemelä, I. 2006. Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence* 170(12): 1031–1080.

Robertson, N.; and Seymour, P. 1991. Graph minors. X. Obstructions to tree-decomposition. *Journal of Combinatorial Theory, Series B* 52(2): 153–190. ISSN 0095-8956.

Streeter, M. J.; and Smith, S. F. 2007. Using Decision Procedures Efficiently for Optimization. In In Proc. *ICAPS*, 312–319.

Suda, M. 2014. Property directed reachability for automated planning. *Journal of Artificial Intelligence Research* 50: 265–319.

Vizel, Y.; Weissenbacher, G.; and Malik, S. 2015. Boolean Satisfiability Solvers and Their Applications in Model Checking. *Proceedings of the IEEE* 103(11): 2021–2035.

Williams, B. C.; and Nayak, P. P. 1997. A reactive planner for a model-based executive. In In Proc. *IJCAI*, volume 97, 1178–1185.