Executing Action Languages for Planning Problems on Multi-core Platforms: Some Preliminary Results

To Thanh Son, Phan Huy Tu, Enrico Pontelli, Tran Cao Son

New Mexico State University
Department of Computer Science
{sto,tphan,epontell,tson}@cs.nmsu.edu

Abstract. The goal of this paper is to demonstrate that parallel programming techniques can boost AI planning systems in various aspects. It shows that an appropriate parallelization of a sequential planning system often brings gain in performance and/or scalability. We start by describing general schemes for parallelizing the construction of a plan. We then discuss the applications of these techniques to two domain-independent heuristic search based planners—a competitive conformant planner (CPA) and a state-of-the-art classical planner (FF). We present experimental results which show that performance improvements and scalability are obtained in both cases. Finally, we discuss the issues that should be taken into consideration when designing a parallel planning system and relate our work to the existing literature.

1 Introduction and Motivation

Planning is the problem of finding a sequence of actions that changes the state of the world from an initial state to a state that satisfies a given set of requirements. Planning is computationally hard. Even the problem of searching for a polynomially bounded plan for propositional domains is NP-complete [3], and it becomes computationally even harder (Σ_P^2-complete) when the initial state is incomplete [3]. These complexity results suggest that domain-independent planners are likely to fail to build a plan within acceptable time bounds for certain problem instances.

In spite of these limitations, over the years, we have witnessed a continuous interest by researchers in building domain-independent planners. The challenge has led to the development of more creative ways to attack this problem, such as the development of new data structure (e.g., the planning graph [4]), the development of several domain-independent heuristics (e.g., [5, 15]), and the development of new planning techniques (e.g., SAT-based or model checking based planners [18, 9], Answer Set planning [20]).

Conformant planning is the problem of finding a sequence of actions that changes the state of the world from *every* possible initial state (or equivalently, a set of initial states) to a state that satisfies a given set of requirements. Like classical planning—which deals with planning problems in presence of a *complete* initial state—conformant planning can be viewed as a *search* problem. Several approaches to conformant planning have been developed. Graphplan is extended in [23] to deal with incompleteness of the initial state. Satisfiability and model-based planning have been applied to conformant planning in [9, 8]. Conformant answer set planning is discussed in [13].

Planning as *heuristic search* has played an important role in planning research. Indeed, heuristic search planners are among the best domain-independent planners[1] for *classical domains* (e.g., FF, HSP2, AltAlt, etc. [1]). For domains with *incomplete* information, heuristic search planners or *conformant planners* are also among the fastest developed (e.g., [6, 7]). The main difference between classical planning and conformant planning lies in the size of the search space. The former searches for solutions in the *state-space*—which can be exponential in the number of propositions in the problem—while the latter performs the search in the *belief-state* space—which is double exponential in the number of propositions.

It is apparent that one of the keys to the success of domain-independent search based planners is the design of good heuristics and a better and compact representation of the search space (e.g., [7]). It is known, however, that there is a trade-off between the cost of computing a heuristic function and its performance. In this context, it is interesting to observe that a domain-independent conformant planner, called CPA [24], can be competitive with many state-of-the-art conformant planners, even though it uses a rather simple heuristic to guide its search. Instead of employing a sophisticated heuristic, CPA uses *approximation* (more on CPA in the next section). The performance of CPA demonstrates that, for various classes of planning problems, the performance of a heuristic function can be compensated by *changes in the reasoning mechanisms*.

In this paper, we continue along the same line of thought, proposing another mechanism, orthogonal to the development of new heuristics, to enhance performance of heuristic search-based planners. More precisely, we investigate the use of *parallel machines* to extract concurrency from the reasoning process employed in planning. Our approach is motivated by: *(a)* the demand for solving larger planning instances; *(b)* the architectural trends of providing users with affordable multi-core platforms; *(c)* the availability of affordable components to build Beowulf clusters; and *(d)* the observation that, with few exceptions (e.g., [26]), existing planning systems are tailored to *sequential computing platforms*.

These issues lead to the following research questions: *(1) Can the computing power of parallel platforms be used to improve planning efficiency and to solve larger problems, which cannot be solved by current planners?* and *(2) Which parallel computing techniques can be applied to planning?* In this paper, we answer these questions.

Our study offers the following outcomes: *(1)* Parallel machines can be effectively used in planning, to solve larger problems and speed up planning time; *(2)* Parallel search techniques [14] cannot be applied blindly; *(3)* Parallel machines can effectively compensate the informativeness of the heuristic function; *(4)* State-of-the-art planners can be adapted to take full advantage of the added computational power provided by multi-core and distributed platforms.

Let us underline that an extensive literature exists dealing with the development of parallel solutions to search problems in various domains (e.g., constraint solving, logic programming, SAT, combinatorial optimization [14, 22, 28, 19]). Although these works provide a basic backbone to our effort, they also clearly highlight that a generic solution

[1] SAT-based planners are becoming more competitive (`zeus.ing.unibs.it/ipc-5`), due to efficient SAT-solvers, but we believe heuristic-based planners will remain important for many years.

to parallel search does not exist, and solutions developed in other domains need to be properly modified to be effective in the context of planning. Thus, it is important to know what will be the pros and cons of parallelization of a planning system and what are the issues that need to be investigated. We hope this paper will provide directions in answering such questions.

2 Preliminaries

2.1 Action Representation and Planning as Search

We employ the action language $\mathcal{AL}$ in [2] for action representation, and we represent a planning problem instance as $\mathcal{P} = \langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$, where $\mathcal{D}$ (the *planning domain*) encodes the actions with their effects and preconditions, and $\mathcal{I}$ and $\mathcal{G}$ describe the *initial state* and the *goal*. $\mathcal{I}$ can be represented by a set of states $\Sigma_{\mathcal{I}}$, and it is said to be *complete* if $|\Sigma_{\mathcal{I}}| = 1$. An action expressed in the *Planning Domain Definition Language (PDDL)*

```
(:action action_name
 :parameters (list of ?xi - typei)
 :precondition (condition)
 :effect (and (when cond1 => eff1) ...))
```

can be viewed as a set of ground laws in $\mathcal{AL}$ as follows. For each valid vector $\boldsymbol{x}$ of the parameters of action_name, we have the set consisting of an executability condition

executable action_name$(\boldsymbol{x})$ **if** condition

and a set of action effect rules

action_name$(\boldsymbol{x})$ **causes** effi **if** condi

for $i = 1, \ldots, k$. The main difference between $\mathcal{AL}$ and PDDL is that $\mathcal{AL}$ allows for the representation of arbitrary axioms. The usefulness of axioms in planning has been discussed in [25].

The semantics of a planning domain can be defined by a *state-transition system* $(\mathcal{S}, \mathcal{A}, \Phi)$, where $\mathcal{S}$ is the set of states, $\mathcal{A}$ is the set of actions, and Φ is a mapping from a pair of the form *(action, state)* to a set of states. For an action a and a state s, $\Phi(a, s)$ denotes the set of possible states resulting from the execution of action a in the state s. When a is not executable in s (or its preconditions are not satisfied in s), $\Phi(a, s) = \emptyset$. An action is *deterministic* if $|\Phi(a, s)| \leq 1$.

For a set of states S, also called a *belief state* (or b-state for short), and an action a, we say that a is executable in S if a is executable in every $s \in S$, and we write $\Phi(a, S) = \bigcup_{s \in S} \Phi(a, s)$; otherwise, $\Phi(a, S) = \emptyset$ (a is not executable in S). Φ is also extended to $\widehat{\Phi}$, which maps action sequences and b-states to b-states. $\widehat{\Phi}(\alpha, S)$ is defined as $\widehat{\Phi}([], S) = S$; and $\widehat{\Phi}([a_0, \ldots, a_n], S) = \widehat{\Phi}([a_1, \ldots, a_n], \Phi(a_0, S))$. An action sequence α is a *solution* to $\mathcal{P}$ (a *plan*) if $\widehat{\Phi}(\alpha, \Sigma_{\mathcal{I}}) \neq \emptyset$ and for every $s \in \widehat{\Phi}(\alpha, \Sigma_{\mathcal{I}})$, s satisfies $\mathcal{G}$.

Here, we experiment with planning problems where: (a) actions are deterministic; (b) the domains might or might not contain axioms; and (c) the initial state might be incomplete.

Figure 1 shows a generic algorithm for planning as search, where $\Sigma_{\mathcal{I}}$ is the initial belief state and $\mathcal{G}$ is the goal (we assume that $\Sigma_{\mathcal{I}}$ does not satisfy $\mathcal{G}$).

```
Algorithm 1: FwdPlan(D,I,G)
1. S = Σ_I; Queue = {(S,[])}; Visited = {S}
2. while Queue is not empty
3.     select (S,p) with the best
         heuristic value from Queue
4.     for each action a executable in S
5.       S' = Φ(a,S)
6.       if S' satisfies G then return [p;a]
7.       else if S' ∉ Visited
8.         compute heuristic for S'
9.         insert (S',[p;a]) into Queue
10.        insert S' into Visited
```

Fig. 1. A heuristic forward search algorithm

2.2 Two Sequential Planning Systems

We use two sequential planning systems in our experiments. The two systems have been chosen because of their efficiency, the fact that they are both search-based planners, and the availability of their source code.

The first system, *FF*,[2] is a planner for classical domains, where the initial state is complete. FF is one of the state-of-the-art classical planers [16], and has received several awards for its outstanding performance in international planning competitions. The input of FF is PDDL. This version of FF does not consider axioms. In FF, the next state is computed by the equation $\Phi(a, s) = s \cup e^+_a(s) \setminus e^-_a(s)$ where $e^+_a(s)$ (resp. $e^-_a(s)$) is the set of positive effects (resp. the set of negative effects) of a in the state s. For this reason, the computation of $\Phi(a, s)$ is very fast as it can be done by two set operations. FF also modifies the general algorithm of Fig. 1 by adding an initial phase of local search (based on hill-climbing), and entering the best-first search only upon failure of the local search phase. The outstanding performance of FF *can be attributed to the accuracy of its heuristic*.

The second system used in our study is a modification of the CPA system[3] [24], developed for conformant planning problems (i.e., $|\Sigma_{\mathcal{I}}| \geq 1$). CPA uses $\mathcal{AL}$ as its input language and the number of fulfilled subgoals as heuristic measure (which is known for being not very accurate). CPA cannot match the performance of FF in most of the classical domains. Nevertheless, CPA is competitive with most of the state-of-the-art conformant planners (at the time it was developed). The main difference between CPA and other conformant planners is that it considers axioms directly, and it uses Φ^*, a deterministic approximation of Φ, to deal with non-determinism (caused by axioms) and incompleteness. Informally, given an action a and a set of literals δ approximating the state of the world, the next state of the world $\Phi^*(a, \delta)$ is approximated by a fixpoint of the process that computes *(i)* the set of fluents that cannot possibly be changed or hold δ_i; and *(ii)* the set of direct effects of a in δ_i where $\delta_0 = \delta$. This process might require n^2 steps, where n is the number of propositions in the domain. Detailed definitions of $\Phi^*(a, \delta)$ can be found in [24]. As we will see later, this will be an important source of parallelism.

[2] See members.deri.at/~joergh/ff.html.
[3] See www.cs.nmsu.edu/~tphan/software.htm

3 The Parallel System — General Schemes

In the rest of the discussion we will use the generic term *computing agents* (or, simply, *agents*) to indicate the concurrent planning engines, concretely implemented as threads or processes. The generic structure of a forward search algorithm, as illustrated in Fig. 1, suggests two natural approaches for the transparent exploitation of parallelism.

3.1 Vertical Parallelism

The algorithm in Fig. 1 explores a search space, described by the possible b-states and the function Φ (or Φ^*). *Vertical Parallelism* arises from the exploitation of parallelism from the non-determinism of the search process—i.e., allowing the concurrent exploration of different (S, p) extracted from the queue (**Line** 3). This is intuitively described in Fig. 2 (left). Effectively, the different agents are exploring alternative ways to reach a goal b-state, by concurrently building distinct plans. The advantage of this approach is the possibility of pursuing alternative plans, which is particularly advantageous when the heuristic function is ineffective in discriminating between b-states to explore. On the other hand, if the different agents use a common representation of the search space (e.g., a single queue), then we run the risk of *(1)* exploring speculative parts of the search space (e.g., b-states with a low heuristic value), and *(2)* modifying the order of exploration of the search space (which might negatively impact the heuristic search).

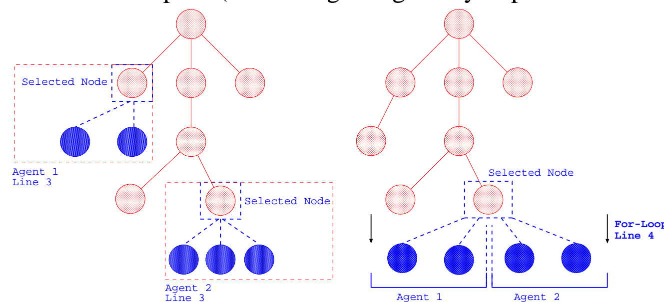

Fig. 2. Vertical and Horizontal Parallelism

3.2 Horizontal Parallelism

Horizontal Parallelism arises from the use of different agents in constructing *one* particular plan—thus, the agents are cooperating in the development of a *single* plan. This is effectively achieved by distributing the iterations of the for-loop of **Line** 4 between the agents, as intuitively illustrated in Fig. 2 (right). The advantage of this approach is the fact that the structure of the search space is unchanged w.r.t. a sequential execution—thus, parallelism does not interfere with the heuristics function. The drawbacks arise from the potentially small granularity of the tasks (e.g., when the states in $\Phi(a, S)$ are "easy" to compute) and the possible contention in the use of a common queue.

4 Systems Description

4.1 mCpA: a Parallel CPA

The parallel versions of CPA have been developed using a common parallel structure. The model adopted relies on a multi-core platform, relying on shared memory for all communication tasks. The agents in charge of performing the computation are represented by concurrent threads; all forms of communication between the computing agents are realized through access and modifications of data structures allocated in the shared memory. Synchronization is required to coordinate access to shared memory (via mutex locks).

We explore alternative implementations, aimed at exploiting specific computational features of the planning domains. VERT is a purely vertical implementation, aimed at domains where the heuristics function is not satisfactory (e.g., many b-states are generated with the same best heuristic value). The horizontal models (HORZ1-HORZ4) are aimed at domains where the computation of the successor states is expensive. In particular, HORZ1 considers the case of large number of actions or actions having comparable "complexity", while HORZ2 and HORZ3 address domains where the number of actions might be large but the cost of applying the actions and determining successor states might widely vary. The HORZ4 prototype considers domains where actions are very simple (not expensive) and overheads should be avoided. The final two models (HYBR1 and HYBR2) address domains where both vertical and horizontal conditions are present (possibly only in moderate terms).

Vertical Parallelism In this implementation, called VERT, we maintain a unique central queue, representing the frontier of the search space, with the b-states ranked according to the heuristics function (as in Fig. 1). The central queue is a priority queue and is allocated in shared memory. We also use a shared table to store visited b-states. Modifications of the central queue and the visited b-states table are critical sections and mutex locks are acquired by the agents for each update.

During the search, each agent P extracts a b-state S with the highest heuristic value from the central queue and determines actions that are executable in S. For each executable action, P computes the successor b-state S' and its heuristic value. If S' satisfies the goal then a solution is returned. Otherwise, if S' is not present in the visited b-states table, P will add S' to the queue and to the visited b-states table. When an agent P satisfies the goal, it sets a flag to notify the others of termination. To detect the situation in which there is no solution, each agent P is associated to a flag that indicates whether P is idle—P is idle if it runs out of work and the central queue is empty. When all agents are idle they stop and report that no solution has been found.

Horizontal Parallelism For horizontal parallelism, we have realized four different implementations—exploring alternative strategies to interact with the central queue.[4]

The HORZ1 implementation relies on the use of a central queue to store open nodes, and a set structure to store visited b-states, both located in shared memory. In HORZ1,

[4] This type of variations have led to significant differences in other parallel systems [14].

we *statically* divide the set of all actions into equal size segments, and assign each segment to a distinct computing agent. During the search, an arbitrary agent, say P_0, extracts a b-state S with the highest heuristic value from the queue, and all agents, including P_0 itself, compute the successor b-states for S. Each agent P computes the successor b-states of S for each applicable action belonging to the segment of actions assigned to P. These b-states are stored locally and only in the end transferred to the central queue (with corresponding update of the visited b-states table)—this guarantees, in most of the cases, that the b-states in the central queue are in the same order as in a sequential execution of CPA.[5]

The HORZ2 implementation is similar to HORZ1, however, instead of assigning to an agent a fixed number of actions to compute the successor b-states, we *dynamically* allocate actions to agents at run time. Given n agents and m actions that need to be tested in computing the successor b-states, if agent P is available, then P will receive a segment containing $\frac{m}{4 \times n}$ unexplored actions[6]; at the same time, the value m of unexplored actions is updated to $m - \frac{m}{4 \times n}$. This process is repeated until the successor b-states for all actions have been computed. The net effect is to start by assigning coarse tasks to the agents, and refining them to smaller tasks as the computation continues, ultimately creating a better load balancing between the computing agents (since different actions may require a different amount of time). All the computed successor b-states are stored in a local queue, associated to each agent. As in HORZ1, when all the actions have been applied, the content of the local queues is transferred to the central queue.

HORZ3 is similar to HORZ2, with the exception that each agent receives only one action at the time—creating fine grained tasks and facilitating load balancing.

Finally, the HORZ4 is identical to HORZ3, except that the b-states computed by the agents are immediately inserted in the central queue. The goal is to avoid the sequential phase required to transfer b-states from the local queues to the central one. The new b-states may appear in the central queue in an order different from the one of sequential execution (if multiple b-states with the same highest heuristics value are present). Thus, the parallel planner is likely to explore the search space in a different order than sequential execution.

Hybrid Parallelism HYBR1 is a combination of horizontal and vertical parallelism. The agents are divided into groups of equal size (4 in our experiments). An arbitrary agent P of each group extracts the b-state from the top of the central queue, and shares the work of computing successor b-states with all agents in the group, including P itself, as done in HORZ1. Checking visited b-states is done as in the previous implementations. The central queue and the visited b-states table are allocated in shared memory.
In HYBR2, agents are divided into two groups—each with a private queue and visited states table. Unlike HYBR1, where all groups play the same role, in HYBR2, the first group uses the strategy of VERT: at any time during the search each agent in the group extracts the b-state from the top of the group's queue, computes the successor b-states for executable actions and inserts them into the group's queue. The second group works in the same way as HORZ1, where actions are divided into segments of equal size and

[5] There are rare exceptions to this, as discussed later.
[6] This formula has been experimentally chosen.

each segment is associated with an agent in the group to compute successor b-states. The intuition is to provide a balance between trusting and speeding up the original heuristics (pursued by the second team), and allowing the fast exploration of alternative branches.

4.2 mFF: a Parallel FF

The parallelization of FF follows the model of vertical parallelism. The choice of not exploring horizontal parallelism is dictated by the high-efficiency of FF in computing Φ, which would make horizontal parallelism too fine-grained. The implementation of vertical parallelism has been adapted to FF with the following main modifications:

- One agent (*master*) maintains the central queue, and performs best-first search following the sequential FF scheme.
- The other agents (*slaves*) are allowed to extract b-states from the central queue and perform search starting from the given b-state. Given a b-state S, the slave agent proceeds by first attempting a hill-climbing local search starting from S, and entering the best-first search only upon failure of hill-climbing (thus, effectively restarting the FF computation from S as initial state). The best-first search conducted by the slave is limited by a maximum number of steps, and the frontier of the final search tree developed by the slave agent replaces S in the central queue (if a solution is found, it will be reported and the computation will terminate).

Asserting a limit on the best-first search conducted by each slave agent is useful to guarantee that the slave agent does not enter a "low quality" part of the search tree, without requiring excessive interaction with the other agents and the central queue. The number of steps allowed is initially set to an experimentally determined constant (500 in the proposed experiments) and it is adaptive—it is incremented as the search goes deeper into the global search tree.

5 Experimental Evaluation

5.1 Benchmarks

To evaluate the performance of our systems, we use two test suites. The first includes classical planning domains, while the second includes conformant planning domains. The benchmarks are briefly described in Table 1. In the discussion, we loosely use the term *speedup* to denote the ratio between the sequential and the parallel execution time, a measure of how much parallel search contributes to improve the performance of a given sequential system.[7] Note that we do not present all the results (e.g., performance of each form of parallelism for each benchmark) due to lack of space.

5.2 Classical Domains

We experimented with both mCpA and mFF on classical planning domains. In both cases, we selected domains and instances which have been proved challenging for the

[7] This is different from the theoretical notion of speedup, that requires the absolute best sequential time using *any* sequential system.

Benchmark	Instances	Source	Brief Description
	Classical Domains		
Gripper	$n = 100$	AIPS 1998	robot transports n balls between 2 locations
Miconic	$p = 20, f = 20$	AIPS 2000	lift transports p passengers between f floors
Pathways	$p = 1, \ldots, 30$	IPC-5	find sequence of biochemical reactions producing p substances
PipesWorld	$p = 10$	IPC-5	Control flow of oil in a p-node network
Stacks	$p = 30$	IPC-5	Problem in production scheduling
Storage	$p = 17$	IPC-5	moving crates from containers to p depots
	Conformant Domains		
Bomb	$b = 200, t = 20$	[6]	disarm bombs by dunking in the toilet
Cleaner	$n = 10, p = 50$	[24]	robot cleans p objects in n rooms
Ring	$n = 30$	[9]	robot locks windows in n rooms
Cube	$n = 9$	[9]	robot moves in a $n \times n \times n$ grid
Safe	$n = 50$	[6]	open a safe with n possible combinations)
Logistic	$l = 3, c = 3, p = 3$	[6]	transport p packages within l locations in c cities
Coin	$n = 10$	IPC-5	robot collects n coins scattered in a building
Comm	$s = 14/12, p = 11/9$	IPC-5	certify or repair packets

Table 1. Benchmarks

sequential counterparts. Times are reported in seconds and the experiments rely on a 2-hour time limit (*TO* denotes time-out). Each problem was solved four times and average execution times are reported.

`mCpA`: `mCpA` has been implemented on a Sun multicore shared memory machine, with 8 cores, 4GB of memory, and running Solaris 9. Table 2 reports the experimental results using different numbers of agents (from 1 to 8). The leftmost column of the table shows the problem and the version of `mCpA` used. In each of the other cells, we report the execution time for the corresponding parallel version, and the corresponding number of agents, followed by the ratio of performance improvement w.r.t. sequential CPA.

VERT obtained good performance on the Miconic domain—a speedup of 12.94 using 8 agents in $Mic(20, 20)$. The reason for the super-linear speedup is due to the fact that multiple agents are exploring different branches of the search tree, and VERT determines a shorter plan than CPA. E.g., for $Mic(20, 20)$, the length of the plan that VERT with 8 agents returned is 86, comparing to 166 of the sequential version. The speedups of HORZ1 and HORZ3 on this problem are also fairly good—more than 6 using 8 agents. They are lower due to the relatively fast computation of successor b-states. We observe drops in speedups in the other systems due to contention on locks and lack of sufficient agents to follow the most promising plans.

In the Gripper domain, the length of the plan returned by VERT is not always shorter than that found by CPA. E.g., with 8 agents, the length of the plan returned by VERT is 377 while that of CPA is 319. That explains why the speedup of the vertical parallel implementation VERT is not as good. In general, the speedups obtained by all the versions, on this domain, are stable, gradually increasing as the number of agents increases. This is because the length of the plan found by the sequential version is not far from that of

an optimal plan (i.e., the heuristics of CPA works well on this domain). Thus, in most of the cases, the parallel solutions represent better approximations of the optimal plan. Note that, in general, optimality is desired but not required.

The Pathway domain, with deterministic actions and a large number of actions and fluents and without axioms, reminds us how important heuristic is in searching for a plan. Here , horizontal parallelism does not pay off, while vertical parallelization provides good speedup.

Domain	n=2	n=4	n=8
Mic(20,20)	CPA: 1232		
VERT	396 (3.12)	193 (6.39)	95 (12.94)
HORZ1	637 (1.94)	340 (3.63)	204 (6.04)
HORZ2	693 (1.78)	525 (2.35)	398 (3.10)
HORZ3	639 (1.93)	333 (3.70)	197 (6.25)
HORZ4	1094 (1.13)	291 (4.23)	154 (7.99)
HYBR1	617 (2.00)	400 (3.08)	182 (6.78)
HYBR2	1236 (1.00)	637 (1.94)	198 (6.21)
Gripper(100)	CPA: 5255		
VERT	3095 (1.70)	2109 (2.49)	1032 (5.09)
HORZ1	2711 (1.94)	1395 (3.77)	778 (6.75)
HORZ2	2821 (1.86)	1499 (3.51)	878 (5.99)
HORZ3	2687 (1.96)	1379 (3.81)	763 (6.89)
HORZ4	2638 (1.99)	1414 (3.72)	788 (6.67)
HYBR1	3899 (1.35)	3260 (1.61)	1591 (3.30)
HYBR2	5276 (1.00)	2722 (1.93)	1404 (3.74)
Pathways(4)	CPA: 23.02		
VERT	4.57 (5.04)	7.2 (3.20)	3.39 (6.79)
HORZ1	20.05 (1.15)	17.51 (1.31)	
HORZ2	15.76 (1.46)	12.24 (1.88)	12.77 (1.80)
HORZ3	15.87 (1.45)	11.03 (2.09)	12.01 (1.92)
HORZ4	15.07 (1.53)	10.06 (2.29)	6.75 (3.41)
HYBR1	12.85 (1.79)	7.12 (3.23)	6.98 (3.30)
HYBR2	27.08 (0.85)	5.96 (3.86)	6.23 (3.70)

Table 2. Classical Benchmarks using `mCpA`

`mFF`: `mFF` has been developed on the same 8-core Sun server, running Solaris 9. The experiments conducted deal with instances of problems (drawn from the IPC-5 competition) that are challenging for the sequential FF system. Table 3 shows the main results. In parentheses we show the ratio between sequential time and parallel time. The important result to underline here is the ability to solve various instances that are intractable by sequential FF (actually, many of the time-outs reported are for instances that took longer than 24 hours). In this context, the benefits are parallelism are two-fold:

- the ability to overlap different *types* of search (standard best-first search and hill-climbing local search)—this is the case of Storage(17), where hill-climbing is time consuming while best-first quickly converges to a solution;

- the ability to concurrently explore branches that have equally high heuristic values (this is the case of Pathways).

Observe that the current implementation is not particularly good in maintaining a low level of communication—we can notice that using 8 agents the performance occasionally degrades (e.g., PipesWorld(15)) due to excessive contentions on the locks of the central queue.

Domain	Instance	Number of Agents			
		FF 2.3	2	4	8
Pathways	9	TO	4.47 (_)	2.71 (_)	3.06 (_)
	11	TO	6.57 (_)	6.55 (_)	3.16 (_)
	12	TO	TO (_)	TO (_)	264.79 (_)
	13	TO	4.64 (_)	3.51 (_)	4.03 (_)
	15	TO	48.39 (_)	4.59 (_)	4.61 (_)
	20	83.08	75.45 (1.10)	21.21 (3.92)	17.83 (4.66)
	30	TO	121.18 (_)	7.09 (_)	7.08 (_)
Stacks	30	202.82	203.16 (0.99)	190.6 (1.06)	185.01 (1.1)
PipesWorld	9	180.64	92.12 (1.96)	67.44 (2.68)	110.01 (1.64)
	10	442.62	48.7 (9.01)	35.28 (12.55)	37.9 (11.68)
	11	64.53	15.49 (4.17)	15.06 (4.28)	12.3 (5.25)
	15	1289.01	1280.15 (1.0)	502.53 (2.57)	1197.8 (1.08)
	20	TO	TO	1393.87 (_)	1259.15 (_)
Storage	17	732.36	8.95 (81.83)	4.93 (148.55)	0.91 (804.4)

Table 3. Results using mFF

5.3 Conformant Domains

The experiments on conformant domains have been performed using mCpA. For reference, we compared our systems with CFF [6] and KACMBP [9], which are two of the fastest conformant planners. Unfortunately, we could not obtain the Solaris executables for these planners, and we had to run them on a Linux machine and scale the timings to our Sun multi-core.[8] The comparison with the other systems is illustrated in Table 4. Some experimental results are shown in Figure 3.

Domain	CPA	CFF	KACMBP
Bomb(200,20)	248	23763	2411
Cleaner(10,50)	390	Maximum length exceeded	TO
Ring(30)	423	TO	< 1
Cube(9)	812	TO	< 1
Safe(50)	1865	68	< 1
Log(3,3,3)	1025	< 1	745
Coin(10)	487	< 1	1502

Table 4. Conformant Benchmarks

[8] For each problem, we compute a time conversion ratio by running CPA on both the Linux and Solaris platforms.

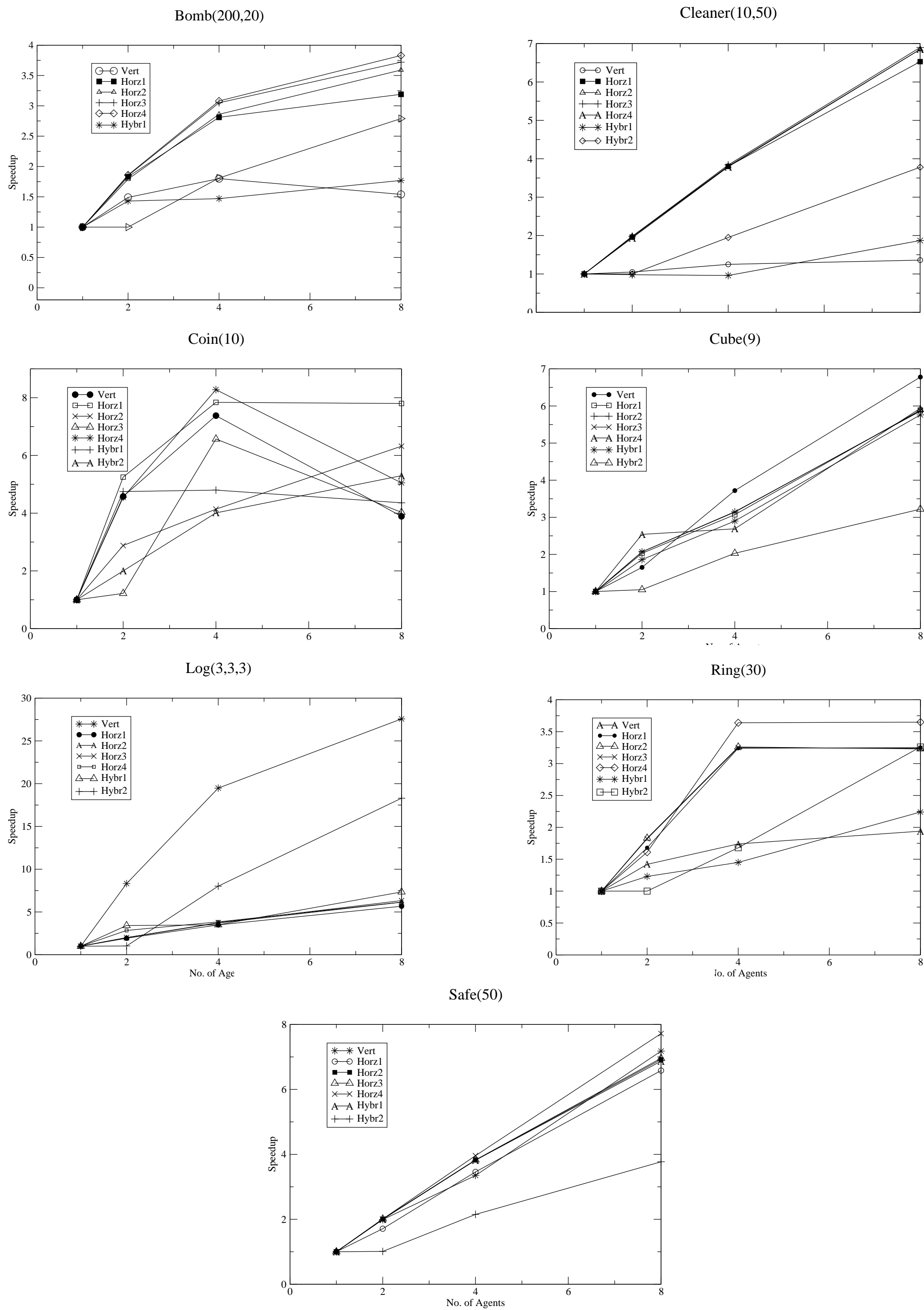

Fig. 3. Speedups on Conformat Domains

In the Bomb and Cleaner domains, we do not see much speedup in VERT when the number of agents increases. This is because the heuristic of CPA works well on this domain. The best parallel implementation on this domain is HORZ4. The reason why HORZ4 is better is because the newly computed successor b-states are immediately inserted in the queue (instead of waiting for all agents to complete). Horizontal parallelism, however, performs very well on the Cleaner domain—since computing successor b-states is expensive.

In the Ring domain, the horizontal parallelism implementations scale well up to 4 agents, and then they become stable. The reason is that the number of actions in the domain is only 4, leaving other agents idle. A similar behavior occurs in the Cube domain, whose number of actions is 6. The speedups of VERT on Cube(9) is good as the system is capable of finding a shorter plan (43 steps, compared to 63 of sequential CPA). In the Safe domain, the speedups of both vertical and horizontal implementations are very good.

The Logistic domain is truly problematic for the sequential version CPA because the heuristic function performs poorly, especially on the Log(4,3,3) problem. Sequential CPA, KACMBP, and most of our parallel implementations, except HORZ2, could not solve it within the time limit. Thanks to parallelism, we could solve this problem (HORZ2) using 8 agents. For the Log(3,3,3) problem, the speedups obtained by VERT and HYBR2 are impressive (more than 25 on 8 agents for the VERT implementation). In the Coin domain, the speedups obtained by our parallel implementations range from 1.22 (HORZ3, 2 agents) to 8.28 (HORZ4, 4 agents).

The occasional super-linear speedups are due to changes in the search pattern caused by parallelism; in the case of vertical parallelism, this is obvious (as multiple paths are concurrently explored). In the case of horizontal parallelism, this may occur because the order of the b-states in the central queue might differ from the sequential execution (the heuristic is currently computed on the first state of a b-state, and this may change during horizontal parallelism).

In summary, on the domains where the heuristic function does not perform well (i.e., various elements receive the highest value, and the one on top of the queue might not lead to the shortest plan), the vertical parallelism is very effective—sometimes we obtain super-linear speedups. In contrast, in the domains where the heuristic function performs well, the speedup obtained by the horizontal approach, although less than linear, is usually good. Furthermore, the more expensive the computation of a successor b-state is, the higher the speedup obtained via horizontal parallelism. The hybrid approach balances these two extremes.

6 Related Work

The work proposed in this paper is in the same spirit as the work of [26], where a parallelization scheme similar to our horizontal parallelism is applied to a STRIPS planner—in their context this is useful to handle the cost of applying operators with variables (while we use it to address the use of axioms), but it will be less effective for domains that have a fast computation of the next state.

Parallel planning as considered in this paper is different from *distributed planning* [21, 12] in that we focus on improving sequential planning systems by distributing the workload of an agent to multiple agents while distributed planning often deals with the problem of coordination between agents to create a plan for all agents. Distributed planning often requires the execution of plans by the agents and agents are often reactive. Agents in our framework share the same goal and representation, stop when one find a plan, and do not execute actions to change the world. Furthermore, efficiency is not the first issue in distributed planning. Issues of parallelization have been explored in this context, by either partitioning actions and goals between agents so that separate plans can be computed and composed (e.g., interaction graphs [17]) or by adopting a hierarchical approach, where distinct "regions" of the plan are given to distinct agents (e.g., [10]). These approaches are "global" versions of horizontal parallelism. Vertical parallelism has been explored in other search-based problems, e.g., [22, 14].

Our approach to planning in this paper is perhaps more closely related to the distributed problem planning in which the planning process is distributed but a centralized plan needs to be found as discussed in [11]. Our proposed approach could be viewed as a special case of system in view of [11] where one agent distributes the work and all agents search for a plan until one is found.

The work presented in this paper is also similar to the path-finding problem in [27] where different search algorithms for finding a path in distributed environment have been proposed. These algorithms do not deal with incomplete information though. Furthermore, we would like to point out that our goal is not to propose a generic algorithms for parallel search.

7 Conclusions and Future Work

Over the years, two main reasons have led to performance improvements of automated planners: new algorithms and faster computers. The latter has allowed to apply the same planning algorithms to solve more complex problems without any changes. This trend is expected to change, as computer manufacturers are moving away from focusing on single-thread performance and focusing on multi-core platforms. In this paper we presented an investigation of alternative methodologies for parallelization of heuristic search-based planners on multi-core platforms. We identified two forms of parallelism and investigated their implementations and interactions. The results are very encouraging, in terms of improved execution time, speedups, and scalability. We are currently exploring the porting of these ideas in a fully distributed platform as well as taking better advantage of the growingly popular hybrid Beowulf clusters—i.e., clusters whose nodes are multi-core platforms.

References

1. F. Bacchus. The AIPS'00 Planning Competition. *AI Magazine*, 22(3), 2001.
2. C. Baral and M. Gelfond. Reasoning agents in dynamic domains. pages 257–279. Kluwer Academic Publishers, 2000.

3. C. Baral et al. Computational complexity of planning and approximate planning in the presence of incompleteness. *AIJ*, 2000.
4. A.L. Blum and M.L. Furst. Fast Planning through Planning Graph Analysis. *AIJ*, 90:281–300, 1997.
5. B. Bonet and H. Geffner. Planning as Heuristic Search. *AIJ*, 129(1–2):5–33, 2001.
6. R. Brafman and J. Hoffmann. Conformant planning via heuristic forward search: A new approach. *ICAPS*, 2004.
7. D. Bryce et al. Planning Graph Heuristics for Belief Space Search. *JAIR*, 26:35–99, 2006.
8. C. Castellini et al. SAT-based Planning in Complex Domains. *Artificial Intelligence*, 147:85–117, July 2003.
9. A. Cimatti et al. Conformant Planning via Symbolic Model Checking and Heuristic Search. *AI Journal*, 159:127–206, 2004.
10. M. desJardins, M. Wolverton. Coordinating Planning Activity and Information Flow in a Distributed Planning System. *AAAI Fall Symp.* 1998.
11. E. Durfee. Distributed Problem Solving and Planning. *Multiagent Systems*, MIT Press. 1999.
12. J. Urban and P. Dasgupta. *The Encyclopedia of Distributed Computing*. Kluwer Pubs., 2003.
13. T. Eiter et al. A Logic Programming Approach to Knowledge State Planning, II. *Artificial Intelligence*, 144(1-2), 2003.
14. G. Gupta et al. Parallel Execution of Prolog Programs: a Survey. *ACM TOPLAS*, 23(4):472–602, 2001.
15. P. Haslum et al. New admissible heuristics for domain-independent planning. In *AAAI*, 1163–1168, 2005.
16. J. Hoffmann and B. Nebel. The FF Planning System: Fast Plan Generation Through Heuristic Search. *JAIR*, 14:253–302, 2001.
17. M. Iwen and A. Mali. Distributed graphplan. *ICTAI*, IEEE, 2002.
18. H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. *AAAI*, pp. 1194–1199, 1996.
19. H. Kitan and J.A. Hendler. *Massive parallel artificial intelligence.* MIT Press, 1994.
20. V. Lifschitz. Answer set programming and plan generation. *AIJ*, 138(1–2), 2002.
21. A.D. Mali and S. Kambhampati. Distributed Planning. *Encyclopedia of Distributed Computing*, Kluwer, 2003.
22. L. Perron. Search Procedures and Parallelism in Constraint Programming. *CP*, Springer Verlag, 346–360, 1999.
23. D. Smith and D. Weld. Conformant graphplan. *AAAI*, 1998.
24. T. C. Son et al. Conformant Planning for Domains with Constraints — A New Approach. In *AAAI*, 1211–1216, 2005.
25. S. Thiebaux, J. Hoffmann, and B. Nebel. In Defense of PDDL Axioms. In *IJCAI*, 2003.
26. D. Vrakas, I. Refanidis, and I. P. Vlahavas. Parallel planning via the distribution of operators. *JETAI*, 13(3):211–226, 2001.
27. M. Yokoo et al. Search Algorithms for Agents. *Multiagent Systems*, MIT Press. 1999.
28. H. Zhang et al. PSATO: a distributed propositional solver and its application. *Journal of Symbolic Computing*, 21(4), 1996.