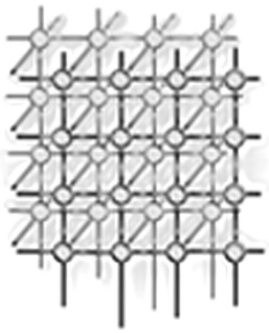
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2000; **00**:1–7 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*



Applications of Parallel Processing Technologies in Heuristic Search Planning: Methodologies and Experiments

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

Department of Computer Science
New Mexico State University
Box 30001, MSC CS
Las Cruces, NM 88003, USA

SUMMARY

The goal of this paper is to investigate the application of parallel programming techniques to boost performance of heuristic search-based 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—on both shared memory and distributed memory platforms—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.

KEY WORDS: Planning; Reasoning about Actions and Change; Application of Parallelism

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

*Correspondence to: Enrico Pontelli, Department of Computer Science, New Mexico State University, Box 30001, MSC CS, Las Cruces, NM 88003, USA, eponteli@cs.nmsu.edu
Contract/grant sponsor: National Science Foundation; contract/grant number:

Received
Revised

hard. Even the problem of searching for a polynomially bounded plan for propositional domains is NP-complete [14], and it becomes computationally even harder (Σ_P^2-complete) when the initial state is incomplete [5]. 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 structures (e.g., the planning graph [7]), the development of several domain-independent heuristics (e.g., [9, 32]), and the development of new computational approaches to planning (e.g., SAT-based or model checking based planners [38, 17, 6, 16], Answer Set planning [46]).

Different classes of planning problems can be identified, depending on the amount and type of knowledge about the initial state of the world that is available to the planning agent. *Complete* knowledge implies that the agent has precise knowledge about *all* the properties of interest describing the initial state of the world. *Incomplete knowledge* implies that the initial values of some properties are unknown. *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 (i.e., the *goal*). This type of planning arises in several application domains, where complete knowledge is an unrealistic expectation and the use of sensing actions might be impractical or too expensive (and used only as a last resource). Conformant planning has gained relevance in recent years, leading also to specialized tracks in the recent planning competitions (e.g., IPC-V). 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 has been extended in [61] to deal with partial observability of the initial state. Satisfiability and model-based planning have been applied to conformant planning in [17, 13]. Conformant answer set planning has been discussed in [25].

Planning as *heuristic search* has played an important role in planning research. Indeed, heuristic search planners are among the best domain-independent planners[†] for *classical domains* (e.g., FF, HSP2, AltAlt, etc. [1, 34, 29]). For domains with *incomplete* information, heuristic search conformant planners are also among the fastest developed (e.g., [11, 12]). 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.

The key 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., [12]). It is known, however, that there is a trade-off between the cost of computing a heuristic function and its performance. In this regard, it is interesting to observe that a domain-independent

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

conformant planner [62], called CpA, has been proved 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*, thus leading to a more compact representation of the search space. 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 this same line of thought, proposing another mechanism, orthogonal to the development of new heuristics, to enhance performance of heuristic search planners. More precisely, we investigate the use of *parallelism* to exploit the concurrency that is inherent in 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;[‡]
(d) the observation that, with few and relatively simple exceptions (e.g., [68]), 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? In turn, can the gain in efficiency be such to improve scalability and solve problems that are beyond the capabilities of existing sequential planning systems?* and *(2) Which parallel computing techniques can be applied to planning?* In this paper, we propose a study of the applicability of different parallel processing techniques to certain significant classes of planning systems—forward-chaining and conformant planners based on heuristic search. Our study offers the following outcomes:

1. Parallel machines can be effectively used to improve the performance of the considered planning systems and enable them to tackle larger problem instances;
2. Parallel search techniques [31] cannot be applied blindly;
3. Parallel machines can effectively compensate the informativeness of the heuristic function;
4. Representative forward-chaining and conformant planners can be adapted to take full advantage of the added computational power provided by both multi-core and distributed memory platforms.

The work proposed in this paper, that we refer to as *parallel planning*, is different from the common notion of *distributed planning* explored in the literature [47, 66]. Our focus is on improving efficiency of sequential planning systems by distributing the workload of a planner to multiple processes, while distributed planning often deals with the problem of coordination between agents to create a plan for all agents.

Let us also underline that an extensive literature exists dealing with the development of parallel solutions to search problems in various domains (e.g., constraint solving, logic

[‡]A Beowulf cluster is simply a cluster of commodity components, usually identical computers interconnected using an off-the-shelf interconnection network (e.g., Myrinet, Infiniband), running open-source software (e.g., Linux, MPI), and dedicated to distributed memory parallel computations.

programming, SAT, combinatorial optimization [31, 54, 71, 39, 44, 43]). These works provide a basic backbone to our effort. On the other hand, it is interesting to observe that the conclusions reached in the context of different domains are radically different; techniques and heuristics found effective for one class of problems (e.g., the use of certain scheduling strategies) have been shown to produce the worst parallel performance in other classes of problems (e.g., as discussed in [43]). This state of affairs is behind the continuously growing literature that explores exploitation of search parallelism from different domains (e.g., the very recent works on parallelization of logic programming, constraint programming, combinatorial optimizations, and theorem proving [52, 44, 20, 51]).

Preliminary experiments conducted using mapping of planning problems to parallel logic programming solutions (e.g., as discussed in [42]) provided very modest parallel performance, suggesting that also the domain of planning may have peculiar behavior in terms of exploitation of parallelism. 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.

The rest of the paper is organized as follows. Section 2 provides some background definitions from the field of planning and reasoning about actions and change. Section 3 introduces the different models of parallelism we explored in our design. Section 4 describes the prototypes developed and their implementation. Section 5 discusses the benchmarks adopted and the results obtained on both multi-core and distributed memory platforms. Section 6 discusses related proposals that have appeared in the literature. Finally, Section 7 presents conclusions and possible future work.

2. PRELIMINARIES

2.1. Action Description Languages

The common practice is to encode a planning domain and its problem instances using languages (*action languages*) with a formal semantics. Action languages describe the domain using two main classes of concepts: *actions*, describing the capabilities of an agent to change the world, and *fluents*, boolean properties used to describe the state of the world. The truth value of fluents may change over time in response to execution of actions. In this project we consider two languages for the description of planning domains: the *Planning Domain Definition Language (PDDL)* [49, 28], for the description of classical domains, and the action language $\mathcal{AL}$ [4], for the description of domains with static causal laws and incomplete knowledge.

2.1.1. The Planning Domain Definition Language (PDDL)

PDDL is a widely used language for the description of planning domains. For example, the planner FF [33] makes use of PDDL as its input language.

Let us summarize some of the key features of basic PDDL—the reader is referred to the literature (e.g., [49, 28]) for a detailed discussion of the syntax and features of this language.

A PDDL definition is composed of two parts: a *domain* description and a *problem* specification.

The domain description defines:

- what predicates are used to describe the properties of the world—known as *fluents* if their state can change over time, otherwise they are known as *timeless*—and
- the actions the described agent can perform.

Both predicates and actions can have parameters. The format of a simple domain definition has the form

```
(define (domain NAME)
    (:predicates   (Pred_Name_1 ?arg1 ... ?argk)
                    ....
                   (Pred_Name_n ?arg1 ... ?argm))
    (:action Action_name_1
           :parameters (?parm1 ... ?parmh)
           :precondition Precond_Formula
           :effect Effect_Formula)
    (:action ... )
)
```

Each action specification describes a class of actions parameterized by the parameters `parm1 ... parmh`, the condition under which they can be executed, and their effects on the world. The precondition could be as simple as an atomic formulae, i.e., `(Pred_Name arg1 ... argk)`, or a more complex formula, such as a conjunction `(and Formula_1 ... Formula_m)`. PDDL allows also the use of disjunctions and quantifiers.

The effect formula could be an atom (which is going to be added as true to the resulting state), a negated atom (which will lead to the deletion of the corresponding atom from the state), or a conjunction of effects. More complex effect formulae (e.g., involving conditional effects and universal quantifiers) are also possible.

A problem definition introduces a description of the initial state (what is true at the beginning of the plan execution) and of the goal—i.e., what properties the final resulting state should satisfy. A typical (simple) problem definition has the form

```
(define (problem Problem_Name)
    (:domain Domain_Name)
    (:objects obj1 ... objk)
    (:init Atom1 ... Atomn)
    (:goal Condition_formula)
)
```

For planning with complete information, the initial state is described by a collection of atoms that are true in the initial state. All the other atoms are assumed to be false. For conformant planning, there have been different uses of PDDL in the specification of the initial states. Some planners use `unknown(l)` to specify that the fluent `l` is unknown. Other uses `oneof(l, not l)` for this purpose. The formula used in the description of the goal has the same form as the formula used for the preconditions of the actions.

The original specification of PDDL allows also the use of *axioms* (a.k.a. *static causal laws*). These are statements of static relationships between propositions within a state. An axiom has the form

```
(:axiom
   :vars (?arg1 ... ?argn)
   :context Condition_formula
   :implies Atom
)
```

It is required that predicates defined by the `:implies` part of an axiom are not allowed to occur as effects of actions. The advantage of axioms in planning has been discussed in [64].

Example 1. *We have a suitcase with two latches l_1 and l_2. Initially, l_1 is up and l_2 is down. To open a latch (l_1 or l_2) we need a corresponding key (k_1 or k_2, respectively). When the two latches are in the up position, the suitcase will be unlocked. When one of the latch is in the down position, the suitcase is locked. The signature of this domain consists of two actions—open(L) and close(L), where $L \in \{l_1, l_2\}$—and three fluents—up(L), locked, and available(K), where $L \in \{l_1, l_2\}$ and $K \in \{k_1, k_2\}$.*

We now present the axioms describing the domain. The open action can be performed only if we have the right key, and the effect is to open the corresponding latch; this can be described as

```
(action open
   :parameters (?latch ?key)
   :precondition (and
                   (key_for ?key ?latch)
                   (available ?key))
   :effect (up ?latch)
)
```

Similarly, the action close will falsify the up property:

```
(action close
   :parameters (?latch)
   :effect (not (up ?latch))
)
```

The fact that the suitcase will be unlocked when the two latches are in the up position, and vice versa, is represented by axioms:

```
(:axiom
   :context (and (up l1) (up l2))
   :implies (not (locked))
)

(:axiom
   :vars ?latch
   :context (not (up ?latch))
   :implies (locked)
)
```

An initial state for this domain could be

```
(:init (key_for k1 l1)
       (key_for k2 l2)
       (locked)
       (available k2)
       (available k1)
       (up l1)
)
```

2.1.2. The Action Language $\mathcal{AL}$

A more expressive language we have considered is the action language $\mathcal{AL}$ [4] for the description of planning domains and planning problem instances in presence of static causal laws and/or incomplete knowledge. A planning problem instance is $\mathcal{P} = \langle \mathcal{D}, \mathcal{I}, \mathcal{G} \rangle$, where $\mathcal{D}$ (the *planning domain*) encodes the actions with their effects and preconditions, and $\mathcal{I}$ and $\mathcal{G}$ describe the *initial state* of the world and the *goal*. The initial configuration of the world $\mathcal{I}$ can be represented by a set of *states* $\Sigma_{\mathcal{I}}$, where a state is an assignment of truth values to the properties of interest (also know as *fluents*). $\Sigma_{\mathcal{I}}$ is said to be *complete* if $|\Sigma_{\mathcal{I}}| = 1$—i.e., there is no ambiguity about the initial configuration of the world.

The description of the planning domain $\mathcal{D}$ is composed of a collection of axioms. Three types of axioms are used:

- For each action `action_name`, we have a set of *executability conditions*
  executable `action_name`$(\vec{x})$ **if** `condition`
  where $\vec{x}$ are the parameters of the action and `condition` is a logical formula. The axiom expresses the conditions the state of the world has to meet to enable the execution of the action.
- For each action `action_name`, we have a set of *dynamic causal laws*
  `action_name`$(\vec{x})$ **causes** `effect` **if** `condition`
  where `effect` is a literal (a fluent or its negation) and `condition` is a logical formula. The axiom asserts that if the action is executed in a state of the world that satisfies `condition`, then it will transform the world in such a way to make `effect` true.
- Axioms (*static causal laws*) of the form
  `effect` **caused_by** `condition`
  where `effect` is a literal and `condition` a logical formula. The axiom states that each state of the world that satisfies `condition` must also satisfy `effect`.

Syntactically, the main difference between $\mathcal{L}$ and PDDL lies in that there is no restriction on static causal laws in $\mathcal{L}$. Therefore, domains in $\mathcal{L}$ can be nondeterministic (i.e., the execution of an action in a state could result into different states), even when actions have deterministic effects. Planning with $\mathcal{L}$ domains is Σ_P^2-complete (see, e.g. [65]).

Example 2. *Let us encode the suitcase domain described earlier using $\mathcal{AL}$. The signature of this domain consists of two actions—open(L) and close(L), where $L \in \{l_1, l_2\}$—and three fluents—up(L), locked, and available(K), where $L \in \{l_1, l_2\}$ and $K \in \{k_1, k_2\}$.*

We now present the axioms describing the domain. Opening a latch puts it into the up position. This is represented by the dynamic causal laws:

$$\begin{array}{l} open(l_1) \textbf{ causes } up(l_1) \textbf{ if } true \\ open(l_2) \textbf{ causes } up(l_2) \textbf{ if } true \end{array}$$

Closing a latch puts it into the down position—this can be written as:

$$\begin{array}{l} close(l_1) \textbf{ causes } \neg\, up(l_1) \textbf{ if } true \\ close(l_2) \textbf{ causes } \neg\, up(l_2) \textbf{ if } true \end{array}$$

We can open the latch only when we have the key. This is expressed by:

$$\begin{array}{l} \textbf{executable } open(l_1) \textbf{ if } available(k_1) \\ \textbf{executable } open(l_2) \textbf{ if } available(k_2) \end{array}$$

The fact that the suitcase will be unlocked when the two latches are in the up position, and vice versa, is represented by the static causal laws:

$$\begin{array}{c} \neg\, locked \textbf{ caused\_by } up(l_1) \wedge up(l_2) \\ locked \textbf{ caused\_by } \neg\, up(l_1) \\ locked \textbf{ caused\_by } \neg\, up(l_2) \end{array}$$

An initial state for this domain could be

$$\mathcal{I} = \left\{ \begin{array}{l} \textbf{initially}(\neg up(l_2)) \\ \textbf{initially}(locked) \\ \textbf{initially}(\neg available(k_1)) \\ \textbf{initially}(available(k_2)) \end{array} \right\}$$

This instance is incomplete, and

$$\Sigma_{\mathcal{I}} = \{\mathcal{I} \cup \{up(l_1)\}, \mathcal{I} \cup \{\neg\, up(l_1)\}\}.$$

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* (or a *plan*) to $\mathcal{P}$ 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:

1. actions are deterministic;
2. the domains might or might not contain static causal laws; and
3. the initial state might be incomplete.

Algorithm 1: FwdPlan($\mathcal{D}, \mathcal{I}, \mathcal{G}$)

```
1.  S = Σ_I ; Queue = {(S, [])} ; Visited = {S}
2.  while Queue ≠ ∅ do
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] endif
7.          else if S' ∉ Visited then
8.              compute heuristic for S'
9.              Queue = Queue ∪ { (S', [p; a]) }
10.             Visited = Visited ∪ { S' }
11.         end if
12.     end for
13. end while
```

Figure 1. A heuristic forward search algorithm

2.2. Planning as Search

Planning can be viewed as a search problem (e.g., [9]) in the space $2^{\mathcal{S}}$ of b-states. 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 $\mathcal{G}$.

The algorithm maintains a queue, which is employed to drive the search process. The items in the queue are pairs, composed of a b-state and the corresponding sequence of actions used to build it. The queue is initialized with the b-state $\Sigma_{\mathcal{I}}$ (i.e., the known initial configuration of the world) and the empty sequence of actions (line 1).

The main loop (lines 2-13) selects the element from the queue with the best heuristic value—i.e., the element (S', ℓ) in the queue such that

$$\text{heuristics}(S') = max\{heuristics(S) \mid (S, l) \in \texttt{Queue}\}$$

The successor states of S' are computed by applying to it all executable actions (line 5). The new states (along with the corresponding sequence of actions) are deposited back into the queue (line 9). Observe that no repeated states are inserted in the queue—thanks to the *Visited* set of b-states (line 7).

The search terminates when a state S' satisfying the goal is constructed (line 6).

2.3. Two Sequential Planning Systems

We use two sequential planning systems in our investigation. The two systems have been chosen because of their efficiency, the fact that they are both search-based planners, and the fact that they address two different classes of planning problems (classical and conformant).

The first system, *FF* (`members.deri.at/~joergh/ff.html`), is a planner for classical domains, where the initial state is complete. FF is one of the state-of-the-art classical planners [33], and has received several awards for its outstanding performance in international planning competitions. This version of FF does not consider static causal laws. 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 relies on two simple 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[§] [62], developed for conformant planning problems (i.e., $|\Sigma_I| > 1$). 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 very competitive with most of the state-of-the-art conformant planners. 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 computation, whose iterations δ_i compute (i) the set of fluents that cannot possibly be changed or hold in δ_i; and (ii) the set of direct effects of a in δ_i—where $\delta_0 = \delta$. This process might require n^2 computations where n is the number of propositions in the domain. Detailed definition of $\Phi^*(a,\delta)$ can be found in [62]. As we will see later, this will be an important source of parallelism.

3. GENERAL SCHEMES FOR PARALLELIZING HEURISTIC SEARCH PLANNERS

We will use the generic term *process* to indicate an instance of the concurrent planning engine that cooperates with other processes in the search for a solution to a planning problem. Observe that this does not necessarily imply that a (Unix) process is used at the implementation level. Indeed, we will make use of threads for the implementations on multi-core platforms. The generic structure of a forward search algorithm, as illustrated in Fig. 1, suggests two natural approaches for the transparent exploitation of parallelism. Intuitively, the two forms of parallelism correspond to the two forms of non-determinism expressed in the algorithm. Using the traditional terminology of logic programming, we have *don't care* non-determinism in Line 4 (computation of the successor states) and *don't know* non-determinism in Line 3 (choice of the next state to explore).

[§] `www.cs.nmsu.edu/~tphan/software.htm`

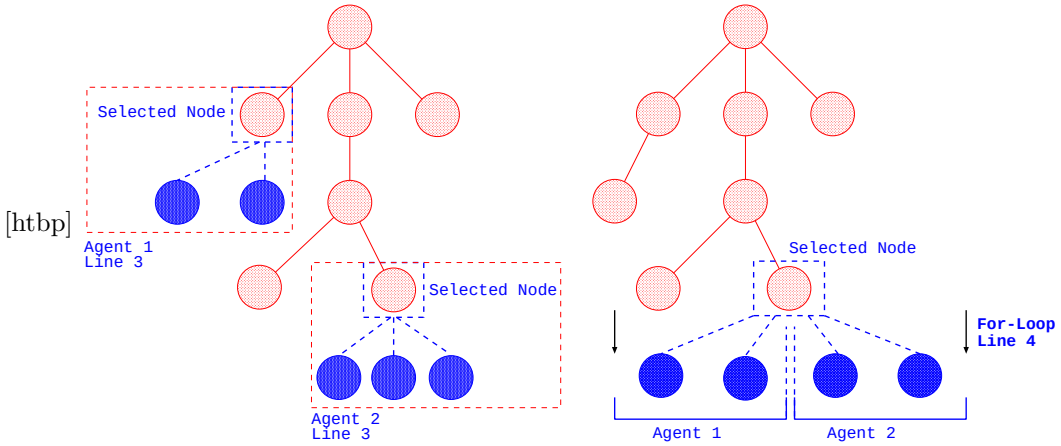


Figure 2. Vertical and Horizontal Parallelism

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. Thus, instead of extracting and exploring only one pair (S, p) from the queue (**Line** 3), we proceed to extract multiple pairs (with high heuristic values) and process them concurrently. This is intuitively illustrated in Figure 2 (left). Effectively, the different processes 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 processes 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.

3.2. Horizontal Parallelism

Horizontal Parallelism arises from the use of different processes in constructing *one* particular plan. Thus, the processes 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 processes, as intuitively illustrated in Figure 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 when depositing the successor b-states computed.

3.3. From the General Idea to a Concrete System

The rest of this paper presents the introduction of these ideas in two concrete planning systems. Before we enter into these details, let us make some general considerations on the possible complications one may encounter.

The key in the exploitation of vertical parallelism lies in the distribution of several states having high heuristic value to different processors for concurrent exploration.

Two essential aspects characterize the way this process is conducted. First of all, states are expected to appear in a representation of a priority queue. The systems we have analyzed provide an explicit priority queue, implemented either using existing libraries (e.g., C++ Standard Template Library) or as an explicitly implemented data structure. The implementation of the priority queue has impact on the ease of parallelization. In a shared memory context (as is the case for multi-core architectures) several threads will concurrently access a common implementation of the priority queue. These accesses have to be treated as critical sections. The naive introduction of mutex locks around the access calls is likely to create severe bottlenecks (as experienced in analogous situations in other domains [67, 44]). A more effective approach requires the use of dedicated parallel data structures to increase the potential of concurrent accesses to the priority queue. Several data structures with such properties have been described in the literature (e.g., [63, 55, 58]).

The representation of the individual states is another critical aspect. In a parallel implementation, states have to be accessed and used by different processors. Certain planners rely on an explicit representation of the components of the state (possibly facilitated by the use of hash tables), while others describe the state in an implicit fashion, e.g., describing it in terms of the sequence of actions required to construct it from the initial state. An implicit representation has the benefit of requiring (on average) less memory, and thus it will reduce the extent of the critical sections when accessing the priority queue. On the other hand, an explicit representation has the advantage of providing a significantly better memory locality.

The choice of which representation of state to rely on is dictated by the underlying parallel architecture. A shared memory implementation suggests the use of explicit state representations; this is particularly true for multi-core platforms, where good locality and good cache behavior is critical (see, e.g., [60]). On the other hand, implementations on distributed memory systems (e.g., Beowulf clusters) suggest the need for reduced communication and short messages. In this context, an implicit representation scheme is preferred.

The exploitation of horizontal parallelism imposes relatively fewer requirements on the existing planning system. The only critical change is related to the ability to distribute sets of actions to different processors for concurrent application. It is fairly common for planning

systems to maintain actions in the forms of indices, which can be easily collected and shared between processors. It is essential that the planner does not rely on excessive global data structures that could complicate the application of each individual action. If information about applicability of each action is collected throughout the execution (in the same style as in planners based on answer set programming [46]), then data structures representing actions will typically have to be duplicated between processors. This is necessary to avoid conflicts. Nevertheless, we have encountered very few systems with this type of problem (e.g., neither FF nor CPA rely on heavy global data structures to determine applicability of actions).

4. SYSTEM DESCRIPTION

4.1. `mCpA`: a Parallel CPA

The parallel versions of CPA have been developed using a common parallel architecture. The model adopted relies on a *shared memory* structure. The processes in charge of performing the computation are represented by concurrent threads—implemented through standard Posix Threads. All forms of communication between the computing processes are realized through access and modifications of data structures allocated in shared memory (global variables). Synchronization is required to coordinate access to shared memory (via Posix 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. The HYBR1 and HYBR2 models address domains where both vertical and horizontal conditions are present (possibly only in moderate terms).

4.1.1. Vertical Parallelism

Our first implementation, called VERT, is based on the vertical parallelism approach.

The implementation of this scheme on a multi-core platform maintains a unique central queue, representing the frontier of the search space, with the b-states ranked according to the heuristics function (as in Figure 1). The central queue is a priority queue, implemented in C++ STL, and it is allocated in shared memory, accessible by all the cores. We also use a shared table (implemented using a C++ STL set structure) 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 processes for each update.

During the search, each individual process P extracts a b-state S with the best heuristic value from the central queue (i.e., the one on top of the queue), and it 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 a process 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 process 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 processes are idle (i.e., their idle flags are set to true) they stop and report that no solution has been found.

The system includes also an efficient implementation of priority queues for handling the central state queue. The scheme adopted is a modification of the algorithm described in [63].

The implementation on a distributed memory platform relies on a similar structure, except that a master process is employed to maintain the central queue. The other processes (slaves) explicitly obtain b-states from the master, perform their computations, and return the new states back to the master for insertion in the queue.

4.1.2. Horizontal Parallelism

For horizontal parallelism, we have realized four systems, exploring alternative strategies to interact with the central queue.[¶] 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 HORZ1 system relies on the use of a central queue to store open nodes—i.e., unexplored b-states—and a set structure to store visited b-states, both located in shared memory. In HORZ1, we *statically* divide the set of all actions into equal size segments, and assign each segment to a distinct process. During the search, an arbitrary process, say P_0, extracts a b-state S with the highest heuristic value from the queue, and all processes, including P_0 itself, compute the successor b-states for S. Each process 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 they are transferred to the central queue only at the end (with corresponding update of the visited b-states table). This structure 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.[‖]

The HORZ2 system modifies HORZ1; instead of assigning to a process a fixed number of actions to compute the successor b-states, we *dynamically* allocate actions to processes at run time. Given n processes and m actions that need to be tested in computing the successor b-states, if process P is available, then P will receive a segment containing $\frac{m}{4\times n}$ unexplored actions.[**] 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 processes, and refining them to smaller tasks as the computation continues, ultimately creating a better load balancing between the processes (since different actions may require a different amount of time to be processed). All the computed successor b-states are stored in a local queue, associated to the corresponding

[¶] This type of variations have led to significant differences in other parallel systems [31].
[‖] There are rare exceptions to this, as discussed later.
[**] This formula has been experimentally chosen.

process. 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 process receives only one action at the time. The objective is to improve load-balancing, by creating very fine grained tasks.

Finally, HORZ4 is identical to HORZ3, except that the b-states computed by the processes 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.

Observe that horizontal parallelism is expected to be fine grained, requiring fairly frequent communications between processes. For this reason, this model of execution seems unsuitable to computing platforms with larger communication latencies, as in the case of Beowulf clusters. Our experiments on horizontal parallelism are limited to the case of multi-core platforms.

4.1.3. Hybrid Parallelism

HYBR1 is a combination of horizontal and vertical parallelism. The processes are divided into groups of equal size (4 in our experiments). At any time, an arbitrary process 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 processes 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 both allocated in shared memory.

The design adopted in HYBR1 has been inspired by the structure of modern Beowulf cluster architectures. These platforms are typically composed of collections of nodes, linked by an interconnection network, where each node is a multi-core machine. Our experiments report implementations of HYBR1 on both a purely shared memory platform (a multi-core machine) as well as cluster, where teams are mapped to nodes and member of each team are mapped to individual cores of a multi-core machine.

In HYBR2, processes 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 process 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 each segment is associated with a process in the group to compute successor b-states. Maintaining the visited b-states table and detecting termination in each group are done as in the other implementations. Intuitively, the design of HYBR2 has been suggested by the desire of making vertical and horizontal parallelism compete against each other. This has been suggested by experimental observations, where certain classes of problems have shown good behavior under vertical parallelism while others have shown poor performance under vertical parallelism but strong results under horizontal parallelism.

4.2. `mFF`: 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 minor modifications:

- One process (*master*) maintains the central queue, and performs best-first search following the sequential FF scheme.
- The other processes (*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 process 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 process 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 process is useful to guarantee that the slave process does not enter a "low quality" part of the search tree, without requiring excessive interaction with the other processes and the central queue. The number of steps allowed is initially set to an experimentally determined constant (500 in our experiments) and it is adaptive. This 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. 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[††]. Note that we do not present all the results (e.g., performance of each system on each benchmark) due to lack of space. In particular, for each problem domain, we present instances that we consider representatives. They are large enough to warrant the use of parallelism and they are representative of the behavior observed on other instances of the domain. Observe that in general we did not observe meaningful differences in the behavior of the planner in different instances of each domain, except for the cases explicitly discussed. We also do not report results in terms of memory consumption; although memory consumption is clearly higher than the sequential case, because of the ability of exploring larger numbers of

[††]This is different from the theoretical notion of speedup [36], that requires the absolute best sequential time using *any* sequential system.

states, in none of the experiments conducted we have observed any memory issue of concern to the feasibility of the executions.

The benchmarks have been drawn from the existing literature and from the international planning competitions, and their main properties are highlighted next.

5.1.1. Classical Domain Benchmarks

Name: Gripper
Source: [50]
Instance reported: $n = 100$
Description: in this problem, a robot needs to transport n balls between two locations. The robot is capable of holding two balls at a time and it can perform three actions (`move`, `pick`, and `drop`). The domain is fairly simple, standard heuristics tend to perform well and produce optimal length plans. The cost for computing the next state is moderate (due to the high number of possible next states).

Name: Miconic
Source: [40]
Instance reported: $p = 20, f = 20$
Description: guide a lift transporting p passengers between f floors; the problem includes constraints on the movement, such as priority of passengers, passengers that must move non-stop, and passengers that have to be accompanied. This domain generates a large space of possible sequences of actions; the actions are simple and the computation of the successor state is fast.

Name: Pathways
Source: [23]
Instance reported: $p = 1, \ldots, 30$
Description: find a sequence of biochemical reactions in an organism to produce selected substances. The parameter indicates the number p of different compounds that have to be created. The creation of compounds require the availability of other substances, that have to be themselves synthesized. Furthermore, different compounds may compete on the use of certain previously created substances. There are 3 different kinds of basic actions corresponding to the different kinds of reactions that appear in the biochemical pathway.

This is a challenging domains; the number of actions and fluent instances is very large. A good heuristics is essential, but often not sufficient to solve larger instances of this domain. In particular, the order in which the goals have to be satisfied is critical. Actions are very simple and the results of their applications can be quickly computed.

Name: PipesWorld
Source: [22]
Instance reported: $p = 9, 10, 15, 20$
Description: Planners control the flow of oil derivatives through a pipeline network obeying various constraints such as product compatibility and tankage restrictions. The parameter denotes the size of the network. The problem generates long plans and large search spaces,

but with simple actions. The problem is very challenging for large instances due to the size of the search space and the ineffective behavior of standard heuristics.

Name: OpenStacks
Source: [22]
Instance reported: $p = 30$
Description: this is an instance of a known combinatorial problem (minimum maximum simultaneous open stacks). A manufacturer has to manage a number of orders, each composed of different products, and each product has to be manufactured in a single instance. The problem is relatively simple, with simple actions and even simple heuristics tend to be effective.

Name: Storage
Source: [22]
Instance reported: $p = 17$
Description: the problem deals with moving a certain number of crates by hoists from containers to depots; within each depot, the hoist can move following given spatial maps. The domain has five actions; the search space is large and even good heuristics tend to be ineffective; the individual actions are simple and can be quickly computed.

5.1.2. Conformant Domains

Name: Bomb
Source: [11]
Instance reported: $b = 200, t = 20$
Description: the problem deals with disarming a number of bombs by dunking them in toilets (that might get clogged). In this domain, the lack of complete initial knowledge leads to a large number of possible worlds; simple heuristics work well in guiding the search, and the individual actions are simple.

Name: Cleaner
Source: [62]
Instance reported: $n = 10, p = 50$
Description: A robot moves around n rooms to clean p objects in each room. The problem has relatively short plans but they require handling a large number of possible worlds. The actions are complex and they activate a large number of static causal laws—making the computation of successor states expensive.

Name: Ring
Source: [17]
Instance reported: $n = 30$
Description: a robot moves around a n-room building to lock windows in a cyclic fashion. This problem has been problematic for many planners. The initial position of the robot as well as the state of the windows is initially unknown. The number of actions is small (4). The large size of the b-states makes the computation of the successor state expensive.

Name: Cube
Source: [17]
Instance reported: $n = 9$

Description: a robot moves in a $n \times n \times n$ grid to reach the center. This is a regular problem, with several actions, a large search space, and hard to tackle with traditional heuristics.

Name: Safe
Source: [11]
Instance reported: $n = 50$
Description: the problem consists of opening a safe whose combination is unknown and there are n possible combinations. The problem has large b-states, which makes the computation of the successor state expensive. The search space is not very large and heuristics tend to be effective.

Name: Logistic
Source: [11]
Instance reported: $l = 3, c = 3, p = 3$
Description: the problem is to transport p packages within l locations in c cities via trucks and airplanes. The uncertainty derives from the unknown location of the package in the origin city. This problem is sensitive to the heuristics used (only the conformant version of FF is capable of solving it quickly). The size of b-states is considerable, but the main complexity is the size of the search space.

Name: Coin
Source: [10]
Instance reported: $n = 10$
Description: a robot needs to pick up n coins randomly scattered on the floors. This is a simpler problem, with a small search space, and where heuristics tend to be effective. The size of the b-states is moderate and there are no static causal laws.

Name: Comm
Source: [10]
Instance reported: $n = 8$ and $n = 10$
Description: several packets need to be certified, and they are replaced if bad. In this problem, we have a large number of initial states, leading to very large b-states. The problem is difficult for the majority of the existing conformant planners.

5.2. Classical Domains

We experimented with both `mCpA` and `mFF` on classical planning domains. In both cases, we selected domains and instances which provide a coverage of different types of domains (in terms of complexity of the search space and numbers of actions and fluents). Times are reported in seconds and the experiments rely on a 2-hour time limit (*TO* denotes time-out). The execution times reported have been obtained as the average execution times from four out of six runs, removing the largest and smallest execution time.

Table I. Classical Benchmarks using `mCpA`

Domain	n=2	n=4	n=8	n=12	n=14
Mic(20,20)			CPA: 1232		
VERT	396 (3.12)	193 (6.39)	95 (12.94)	57 (21.51)	45 (27.54)
HORZ1	637 (1.94)	340 (3.63)	204 (6.04)	161 (7.66)	150 (8.23)
HORZ2	693 (1.78)	525 (2.35)	398 (3.10)	294 (4.20)	265 (4.65)
HORZ3	639 (1.93)	333 (3.70)	197 (6.25)	154 (8.01)	144 (8.57)
HORZ4	1094 (1.13)	291 (4.23)	154 (7.99)	90 (13.66)	200 (6.15)
HYBR1	617 (2.00)	400 (3.08)	182 (6.78)	89 (13.83)	231 (5.34)
HYBR2	1236 (1.00)	637 (1.94)	198 (6.21)	257 (4.80)	220 (5.59)
Gripper(100)			CPA: 5255		
VERT	3095 (1.70)	2109 (2.49)	1032 (5.09)	845 (6.22)	727 (7.23)
HORZ1	2711 (1.94)	1395 (3.77)	778 (6.75)	578 (9.09)	530 (9.92)
HORZ2	2821 (1.86)	1499 (3.51)	878 (5.99)	619 (8.50)	565 (9.31)
HORZ3	2687 (1.96)	1379 (3.81)	763 (6.89)	563 (9.34)	511 (10.29)
HORZ4	2638 (1.99)	1414 (3.72)	788 (6.67)	574 (9.16)	508 (10.34)
HYBR1	3899 (1.35)	3260 (1.61)	1591 (3.30)	1197 (4.39)	901 (5.83)
HYBR2	5276 (1.00)	2722 (1.93)	1404 (3.74)	987 (5.32)	873 (6.02)
Pathways(1)			CPA: 4.2		
VERT	1.79(2.35)	0.56(7.50)	1.49(2.82)	2.92(1.44)	4.12(1.02)
HORZ1	3.76(1.12)				
HORZ2	4.39(0.96)	3.7(1.14)	4.45(0.94)	4.96(0.85)	5.07(0.83)
HORZ3	4.42(0.95)	3.54(1.19)	4.32(0.97)	4.82(0.87)	5(0.84)
HORZ4	4.07(1.03)	2.67(1.57)	0.9(4.67)	3.22(1.30)	3.62(1.16)
HYBR1	4.34(0.97)	3.65(1.15)	1.25(3.36)	4.85(0.87)	0.82(5.12)
HYBR2	6.62(0.63)	1.89(2.22)	1.28(3.28)	2.67(1.57)	1.89(2.22)
Pathways(2)			CPA: 2629.91		
VERT	317.15 (8.29)	96.77 (27.18)	TO (-)	768.49 (3.42)	1210.58 (2.17)
HORZ1	TO (-)	TO (-)	TO (-)	TO (-)	TO (-)
HORZ2	TO (-)	TO (-)	TO (-)	TO (-)	TO (-)
HORZ3	TO (-)	TO (-)	TO (-)	TO (-)	TO (-)
HORZ4	TO (-)	TO (-)	TO (-)	1147.45 (2.29)	TO (-)
HYBR1	3.73 (705.07)	106.25 (24.75)	4.78 (550.19)	88.51 (29.71)	4.66 (564.36)
HYBR2	TO (-)	77.78 (33.81)	1686.58 (1.56)	T0 (-)	T0 (-)
Pathways(4)			CPA: 23.02		
VERT	4.57 (5.04)	7.2 (3.20)	3.39 (6.79)	4.87 (4.73)	7.46 (3.09)
HORZ1	20.05 (1.15)	17.51 (1.31)		3.74 (6.16)	
HORZ2	15.76 (1.46)	12.24 (1.88)	12.77 (1.80)	13.48 (1.71)	14.57 (1.58)
HORZ3	15.87 (1.45)	11.03 (2.09)	12.01 (1.92)	13.15 (1.75)	13.82 (1.67)
HORZ4	15.07 (1.53)	10.06 (2.29)	6.75 (3.41)	2.49 (9.24)	10.59 (2.17)
HYBR1	12.85 (1.79)	7.12 (3.23)	6.98 (3.30)	11.05 (2.08)	7.05 (3.27)
HYBR2	27.08 (0.85)	5.96 (3.86)	6.23 (3.70)	2.17 (10.61)	8.8 (2.62)

5.2.1. `mCpA`

`mCpA` has been implemented on a Sun Enterprise 4500 shared memory machine, with 14 processors, 4GB of memory, and running Solaris 9. Table I reports the experimental results using different numbers of processes (from 1 to 14). 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 processes, followed in parentheses by the ratio of performance improvement w.r.t. sequential CPA.

VERT obtained good performance on the Miconic domain—a speedup of 27.54 using 14 processes in $Mic(20, 20)$. The reason for the super-linear speedup is due to the fact that multiple processes 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 14 processes returned is 86, comparing to 166 of the sequential version. The speedups of HORZ1 and HORZ3 on this problem are also quite good—more than 8 using 14 processes. 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 processes to follow the most promising plans (e.g., the length of the plan doubles in HYBR2 going from 12 to 14 processes).

In the Gripper domain, the length of the plan returned by VERT is not always shorter than that found by CPA. E.g., with 14 processes, the length of the plan returned by VERT is 377 while that of CPA is 319. This explains why the speedup of the vertical parallel implementation VERT is not as good. The speedups obtained by all the versions, on this domain, grow steadily, gradually increasing as the number of processes increases. This is because the length of the plan found by the sequential version is almost optimal (i.e., the heuristics of CPA works well in this case).

The Pathway domain reminds us how important the heuristic is in search. The actions are deterministic and the number of actions and fluents is large. The domain does not contain axioms. As a result, horizontal parallelism does not pay off, while vertical parallelization provides good speedup.

Overall, it is interesting to observe that in the large majority of experiments, parallelism has enhanced the efficiency of planning. We would like to point out that this is not a trivial result, considering the risk of slowdown due to overhead. Thus, our approaches meet the *no-slowdown* criteria, as stated in the literature on parallel logic programming [31]—in general, we have rarely encountered overheads higher than 5% w.r.t. the sequential execution time.

In some benchmarks, we notice an irregular behavior in the execution time when increasing the number of agents. This is not an uncommon behavior and it has been observed in other investigations of parallelization of search (e.g., [31, 15, 54, 45, 30]. This is due to a combination of effects, ranging from additional contention on shared resources (e.g., locks), to the interaction between concurrent search and quality of the heuristic function. In cases like Pathway, the addition of more processes leads to an increased rate of production of nodes on the queue (not effectively distinguished by the heuristic function), pushing the nodes that can lead to the solution away from the top of the queue. The decrease in performance in cases like horizontal parallelism in Pathway is also due to the limited number of applicable actions (for smaller instances) and the heavy contention on the queue—as applying actions is a very fast operation.

Only in the Pathway domain the `mCpA` was able to meet the additional goal of improved scalability—by actually solving instances that were beyond the reach of the sequential version. This is partly due to the overly simplistic structure of CPA, which is really designed to meet the needs of conformant planning and is not very competitive as a classical planner.

5.2.2. `mFF`

`mFF` has been developed on a 6-core Sun T1000, 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 II shows the main results. In parentheses we show the ratio between sequential time and parallel time.

The application of parallelism to FF demonstrates the ability of not only improving efficiency but also enhancing scalability. The important result to underline here is the ability to solve various instances that are unsolvable by sequential FF (actually, many of the time-out reported

Table II. Results using mFF

Domain	Instance	Number of Procs			
		FF 2.3	2	4	6
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 (_)	2.03 (_)
	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)	115.01 (1.76)
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)	17.9 (24.7)
	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)	397.8 (3.25)
	20	TO	TO	1393.87 (_)	1259.15 (_)
Storage	17	732.36	8.95 (81.83)	4.93 (148.55)	0.91 (804.4)

are for instances that took longer than 24 hours). In this context, the benefits of 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 6 processes the performance occasionally degrades (e.g., PipesWorld(9)) due to excessive contention on the locks of the central queue. A more careful implementation using known techniques (e.g., the splitting mechanisms described in [56]) can avoid such issues.

5.3. Conformant Domains

The experiments on conformant domains have been performed using mCpA. For reference, we compared our systems with CFF [11] and KACMBP [17], 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 4500. For each problem, we compute a time conversion ratio by running CpA on both the Linux and Solaris platforms. Some experimental results are shown in Table III. The execution times of the sequential planners are shown in the same row with the problem name. In other cells, $t\,(s)$ stands for the time of the parallel version and the speedup w.r.t. sequential CpA.

In the Bomb and Cleaner domains, we do not see much speedup in Vert, especially in the case of Bomb(200,20), when the number of processes increases. This is because the heuristic of CpA works well on this domain. CpA took 744 seconds to solve Bomb(200,20), whereas

CFF took around one hour and fortyfive minutes while KACMBP timed-out. The best parallel implementation on this domain is HORZ4, where the speedup obtained on Bomb(200,20) peaks at 5.29 using 12 processes. The superiority of HORZ4 is due to the fact that the computed successor b-states are immediately inserted in the queue (without waiting for all processes to complete). The slight decrease of performance at 14 processes is simply due to additional contention on the shared queue.

Horizontal parallelism performs very well on the Cleaner domain, since computing successor b-states is expensive. The speedups obtained by HORZ2—HORZ4 are more than 8 using 14 processes. Vertical parallelism is ineffective and produces a stable speedup of about 1.4.

In the Ring domain, the horizontal parallelism implementations (HORZ1–HORZ4) scale well up to 4 processes, and then they become almost constant. The reason is that the number of actions in the domain is only 4, leaving other processes idle. A similar behavior occurs in the Cube domain, whose number of actions is 6. However, the performance of HYBR1 on Cube(9) is better (9.86 speedup using 12 processes). This is due to the poor performance of the heuristic function. This becomes more apparent when we look at the performance of VERT. The speedups of VERT on Cube(9) using 14 processes is 13.31, and the length of the plan is 43, compared to 63 of sequential CPA. In the Safe domain, the speedups of all systems are very good.

The Logistic domain is problematic for sequential CPA, because the heuristic function performs poorly, especially on Log(4,3,3). 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 10 processes (in 1282 sec.). For the Log(3,3,3) problem, the speedups obtained by VERT and HYBR2 are impressive: the speedup of VERT with 14 processes is 24.82 and the speedup of HYBR2 is 37.56 with 12 processes. The speedups obtained by the other implementations are also good, ranging from 7.40 (HORZ1) to 9.75 (HORZ4). In the Coin domain, the speedups obtained by our parallel implementations range from 1.22 (HORZ3, 2 processes) to 8.28 (HORZ4, 4 processes). In the `Comm` domain, the number of initial states is huge. This forces CPA to search for plans from a b-state consisting of 2^n partial states, leading to time-out for $n > 7$. Most of the parallel versions can solve $Comm(8)$ within 20 minutes. Also, VERT can solve the $Comm(10)$ problem.

The 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, in the domains where the heuristic function does not function well (i.e., various elements receive the highest value, and the one on top of the queue might not lead to the shortest plan), vertical parallelism is very effective, sometimes leading to super-linear speedups. On the contrary, 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 provides a good balance between these two situations.

In general, in the conformant planning problems we can observe significant improvements in efficiency. For example, several instances provide super-linear speedups. We can also observe enhanced scalability in at least one domain (Comm), where problems beyond the reach of the sequential planner can be addressed in a reasonable amount of time. It is also important to observe that in most of the tests the granularity of the parallel work appears to be coarser than what we observed in the non-conformant problems. This leads to better speedups, and the cases where an increased number of processes leads to a reduced speedups are significantly more rare.

5.4. Planning on Distributed Memory Machines

As mentioned earlier, large scale scalability cannot be completely measured on shared memory platforms (e.g., multi-core), as these are currently limited on the number of cores available (e.g., 16-core cpus have been announced for 2008). We want to explore how the proposed models perform on more scalable platforms, specifically Beowulf clusters.

5.4.1. Scalable Engines

We developed two prototypes (MPI1 and MPI2), designed to avoid the use of shared memory and suitable to distributed memory systems.

MPI1 is based on the master-slave paradigm. A *master* process is responsible for controlling the search and distributing the work to all the other processes (the *slaves*). The list of open b-states is stored in a queue in the master, and whenever a slave P is free, the master picks up a b-state with the highest heuristic value from the queue and sends it to P to explore. When a slave runs out of work, it requests a new b-state from the master. For each action a, P computes the successor b-state S' of S, i.e., $\Phi^*(a, S)$. If S' satisfies the goal, then it will return a solution and inform the master by sending a `SOLUTION_MSG` message. Otherwise, it immediately sends S' to the master, to be inserted into the central queue, and continues computing the successor b-state for the next action. In addition to controlling the search and distributing the work, the master also participates in the search whenever there are no incoming messages. To reduce communication between the master and slaves, each process maintains a data structure to store the (locally) visited b-states, and successor b-states are sent to the master only if they are not in such table, which is updated when a b-state is computed by the slave.

MPI2is a combination of the shared memory and distributed memory methodologies. Similarly to MPI1, MPI2 has a special process called the *master* which is responsible for controlling the search and distributing the work. However, each slave (one node of our cluster) is a group of processes instead of a single one. Processes in each slave share the work of computing the successor b-states, as in HYBR1.

5.4.2. Experimental Results

The prototypes MPI1 and MPI2 have been developed on a Beowulf cluster, having 32 dual-processor nodes. We have experimented with problems from the blocks world domain (Block(4)

Table III. Conformant Benchmarks: Time in seconds (speedups in parenthesis)

Domain	n=2	n=4	n=8	n=12	n=14
Bomb(200,20)	CPA: 744		CFF: 71289	KACMBP: 7233	
VERT	499 (1.49)	414 (1.80)	484 (1.54)	634 (1.17)	675 (1.10)
HORZ1	407 (1.83)	265 (2.81)	233 (3.19)	263 (2.83)	276 (2.70)
HORZ2	416 (1.79)	260 (2.86)	207 (3.59)	229 (3.25)	245 (3.04)
HORZ3	402 (1.85)	244 (3.05)	200 (3.72)	233 (3.20)	250 (2.98)
HORZ4	401 (1.86)	242 (3.08)	194 (3.83)	141 (5.29)	151 (4.92)
HYBR1	521 (1.43)	507 (1.47)	421 (1.77)	408 (1.82)	455 (1.64)
HYBR2	745 (1.00)	410 (1.81)	267 (2.79)	235 (3.17)	234 (3.17)
Cleaner(10,50)	CPA: 1171	CFF: Maximum length exceeded			KACMBP: TO
VERT	1110 (1.05)	938 (1.25)	859 (1.36)	823 (1.42)	834 (1.40)
HORZ1	598 (1.96)	308 (3.80)	179 (6.53)	170 (6.89)	173 (6.77)
HORZ2	593 (1.98)	307 (3.81)	171 (6.85)	143 (8.20)	143 (8.18)
HORZ3	592 (1.98)	305 (3.84)	170 (6.90)	143 (8.21)	142 (8.22)
HORZ4	603 (1.94)	309 (3.79)	171 (6.85)	142 (8.27)	141 (8.29)
HYBR1	1194 (0.98)	1216 (0.96)	627 (1.87)	532 (2.20)	483 (2.42)
HYBR2	1172 (1.00)	605 (1.95)	308 (3.78)	226 (5.26)	202 (5.79)
Ring(30)	CPA: 1269		CFF: TO	KACMBP: 1	
VERT	897 (1.42)	728 (1.74)	655 (1.94)	684 (1.86)	668 (1.90)
HORZ1	754 (1.68)	392 (3.24)	391 (3.25)	392 (3.24)	394 (3.22)
HORZ2	695 (1.83)	390 (3.26)	392 (3.23)	392 (3.24)	396 (3.21)
HORZ3	696 (1.82)	390 (3.25)	391 (3.24)	392 (3.24)	394 (3.22)
HORZ4	786 (1.61)	349 (3.64)	348 (3.65)	342 (3.71)	399 (3.18)
HYBR1	1029 (1.23)	873 (1.45)	566 (2.24)	396 (3.21)	330 (3.85)
HYBR2	1271 (1.00)	757 (1.68)	390 (3.26)	390 (3.26)	390 (3.26)
Cube(9)	CPA: 2475		CFF: TO	KACMBP: < 1	
VERT	1502 (1.65)	666 (3.72)	365 (6.78)	240 (10.31)	186 (13.31)
HORZ1	1217 (2.03)	807 (3.07)	420 (5.89)	421 (5.87)	426 (5.81)
HORZ2	1198 (2.07)	787 (3.15)	422 (5.86)	423 (5.85)	427 (5.80)
HORZ3	1195 (2.07)	787 (3.14)	422 (5.87)	422 (5.87)	425 (5.82)
HORZ4	976 (2.54)	919 (2.69)	417 (5.94)	556 (4.45)	381 (6.49)
HYBR1	1332 (1.86)	854 (2.90)	430 (5.76)	251 (9.86)	273 (9.07)
HYBR2	2356 (1.05)	1221 (2.03)	768 (3.22)	425 (5.83)	399 (6.20)
Safe(50)	CPA: 5581		CFF: 232	KACMBP: < 1	
VERT	2808 (1.99)	1667 (3.35)	777 (7.18)	874 (6.39)	722 (7.73)
HORZ1	3257 (1.71)	1611 (3.46)	849 (6.58)	606 (9.20)	598 (9.33)
HORZ2	2785 (2.00)	1461 (3.82)	806 (6.92)	562 (9.92)	550 (10.16)
HORZ3	2778 (2.01)	1456 (3.83)	802 (6.96)	569 (9.81)	508 (10.98)
HORZ4	2778 (2.01)	1410 (3.96)	723 (7.72)	533 (10.47)	516 (10.81)
HYBR1	2785 (2.00)	1461 (3.82)	813 (6.87)	572 (9.76)	514 (10.86)
HYBR2	5497 (1.02)	2595 (2.15)	1488 (3.75)	1388 (4.02)	1208 (4.62)
Log(3,3,3)	CPA: 3510		CFF: < 1	KACMBP: 2197	
VERT	422 (8.32)	180 (19.48)	127 (27.57)	175 (20.07)	141 (24.82)
HORZ1	1815 (1.93)	1005 (3.49)	619 (5.67)	521 (6.73)	474 (7.40)
HORZ2	1760 (1.99)	945 (3.71)	571 (6.14)	461 (7.61)	439 (7.99)
HORZ3	1748 (2.01)	936 (3.75)	555 (6.33)	459 (7.65)	437 (8.03)
HORZ4	1234 (2.84)	915 (3.84)	570 (6.15)	504 (6.96)	360 (9.75)
HYBR1	1029 (3.41)	1003 (3.50)	478 (7.34)	429 (8.18)	387 (9.08)
HYBR2	3408 (1.03)	438 (8.02)	192 (18.30)	93 (37.56)	108 (32.38)
Coin(10)	CPA: 1257		CFF: < 1	KACMBP: 4540	
VERT	274 (4.58)	170 (7.38)	323 (3.89)	326 (3.86)	380 (3.31)
HORZ1	239 (5.25)	160 (7.84)	161 (7.8)	168 (7.46)	234 (5.37)
HORZ2	436 (2.88)	303 (4.14)	199 (6.32)	184 (6.82)	188 (6.7)
HORZ3	1032 (1.22)	191 (6.57)	312 (4.03)	309 (4.07)	249 (5.05)
HORZ4	276 (4.55)	152 (8.28)	249 (5.05)	153 (8.24)	224 (5.6)
HYBR2	265 (4.75)	262 (4.8)	288 (4.36)	341 (3.69)	421 (2.98)
HYBR1	630 (2.00)	313 (4.01)	237 (5.30)	221 (5.70)	224 (5.61)
Comm(8)	CPA: TO		CFF: < 1	KACMBP: TO	
VERT	TO	228.99 (-)	355.95 (-)	433.65 (-)	521.81 (-)
HORZ2	TO	209.78 (-)	188.69 (-)	726.52 (-)	159.55 (-)
HORZ3	TO	183.64 (-)	370.05 (-)	217.07 (-)	180.81 (-)
HORZ4	TO	309.06 (-)	687.38 (-)	158.4 (-)	158.29 (-)
HYBR1	TO	228.26 (-)	247.66 (-)	249.51 (-)	190.51 (-)
HYBR2	TO	211.99 (-)	152.68 (-)	232.99 (-)	294.74 (-)
Comm(10)	CPA: TO		CFF: < 1	KACMBP: TO	
VERT	TO	1541.88 (-)	2000.77 (-)	2917.47 (-)	2744.72 (-)

Table IV. MPI1 and MPI2: Execution Times (in seconds) and Speedups (within parenthesis)

MPI1:	n=1	n=2	n=4	n=8	n=16	n=32
Block(4)	149	84 (1.8)	52 (2.8)	15 (10.3)	14 (10.7)	7 (19.9)
Block(5)	309	92 (3.4)	42 (7.3)	21 (14.6)	21 (15.1)	10 (31.1)
Mic(20,20)	260	52 (5.0)	41 (6.3)	20 (12.9)	14 (18.3)	11 (23.3)
Mic(40,40)	3166	3381 (1.0)	1797 (1.8)	695 (4.7)	384 (8.5)	274 (11.9)
Log(3,3,3)	493	76 (6.5)	37 (13.5)	23 (21.7)	16 (30.0)	14 (36.0)
Log(4,3,3)	3866	1506 (2.57)	797 (4.85)	627 (6.17)	494 (7.82)	265 (14.60)
MPI2:	n=1	n=2	n=4	n=8	n=16	n=30
Block(4)	149	87 (1.7)	41 (3.6)	8 (19.8)	5 (28.4)	7 (21.5)
Block(5)	309	86 (3.6)	23 (13.7)	12 (26.3)	9 (34.4)	11 (27.1)
Mic(20,20)	260	162 (1.6)	31 (8.3)	13 (19.9)	8 (33.2)	9 (29.0)
Mic(40,40)	3166	3965 (0.8)	643 (4.9)	474 (6.7)	160 (19.8)	113 (28.1)
Log(3,3,3)	493	295 (1.7)	28 (17.4)	17 (29.9)	17 (29.6)	27 (18.6)
Log(4,3,3)	3866	3307 (1.17)	1646 (2.35)	575 (6.73)	376 (10.28)	838 (4.61)

and Block(5) from [25]), the Miconic domain (Mic(20,20), Mic(40,40)), and the logistic domain (Log(3,3,3) and Log(4,3,3)).

The experimental results are shown in Table IV. The table reports the execution times and the speedups obtained for each distributed prototype, where n is the number of master/slaves ("n=1" is sequential CPA).

In spite of the relative simplicity of these prototypes, the results are impressive. On the Block(5), the speedups obtained by MPI1 and MPI2, when run on 32 master/slaves, are 31.14 and 27.05. This benchmark has a large search space, with several successful plans. Vertical parallelism is very effective and it is the reason for the high speedups in both engines.

MPI1 also performs well on the logistic domain. The speedups of MPI1 on Log(3,3,3) reaches 35.99 on 32 nodes. The solutions scale well (increased speedups with increase of number of processes). The performance of MPI2 is very good up to 16 slaves. There are occasional drops in speedups for large numbers of slaves, due to excessive traffic in the master. This (fairly infrequent) problem suggests the need to consider either more distributed representations of the queues of tasks or fully distributed scheduling strategies (e.g., as adopted in several implementations of parallel logic programming systems [44]).

The experiments on the Miconic benchmarks are also very satisfactory. This benchmark has a very large search space, but in addition the cost of computation of successor states is relevant. This effect can be observed in Mic(20,20), where better speedups are produced by MPI2, thanks to the ability of each team to use local horizontal parallelism to quickly compute successor states and produce more quickly good states to be distributed to the rest of the system. The same behavior is also demonstrated in Mic(40,40).

6. RELATED WORK

In this work, we provide a uniform characterization of strategies for exploitation of parallelism from heuristic search planning. The idea of exploiting parallelism in planning is not new,

and instances of the individual forms of parallelism (horizontal, vertical) can be found in the literature—though there have not been previous attempts to experimentally compare them.

A form of horizontal parallelism in a heuristic search planner has been described in the *ODMP* project [68]. In ODMP, parallelism is exploited by enabling different agents to concurrently apply distinct actions to a selected state and compute the set of successor states (along with their heuristic values). ODMP works with a simple STRIPS-based planner, in a situation of complete knowledge and without static causal laws). The effectiveness of ODMP arises from those domains where actions have variables and the planner has to invest time to compute action instantiation. The ODMP approach relies on some key assumptions: the planner makes use of an effective heuristics, the number of actions is small and actions are relatively expensive, and the communication costs are low (ideally, the model is designed for a multi-threaded architecture). In this paper, we propose a more general perspective, not limited to these assumptions; in particular, we debate (and experimentally demonstrate) that parallelism can be effective in addressing issues of larger scale scalability and ineffective heuristics.

An example of exploitation of vertical parallelism can be found in the work of Kabanza et al. [37]. In this work, a distributed version of the hierarchical task planning system SHOP is presented (DSHOP). The parallel model described resembles more closely the traditional models of *or-parallelism* explored in logic programming [31]; a master process implements scheduling, by maintaining the central queue of unexplored states, while each slave process completely explores a branch of the search tree, returning new states to the master. Following the tradition of work on or-parallelism, the method explores the distinction between *copying* and *recomputation* to address the sharing of work between processes [43]. The model of vertical parallelism we propose targets heuristic search. It is unclear how to easily adapt the or-parallel search of DSHOP to address heuristics (without excessively increasing interaction between the master and the slaves). Concerning the task sharing strategy, we rely on a copying scheme; the presence of static causal laws suggests avoiding recomputation (which can be very expensive); furthermore, implementations like CPA already include compact closed representations of b-states, that can be easily shared and copied.

As already mentioned earlier, our view of parallel planning, as considered in this paper, is actually different from the notion of distributed planning introduced by several authors [47, 66]. The main difference is that our focus is strictly on improving efficiency of sequential planning systems, by distributing the workload of a planner across multiple processes, while distributed planning often deals with the problem of coordination between agents to create a plan for all agents. A number of distributed planning systems[‡‡] make use of on-line planning, where planning is dynamically inter-mixed with execution steps. Our focus is on off-line planning, where processes share the same goal and representation, and they stop when one finds a plan. 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 [35]) or by

[‡‡]Note that there are also distributed planning systems that are not reactive.

adopting a hierarchical approach, where distinct "regions" of the plan are given to distinct processes (e.g., [21, 19]). These approaches are "global" versions of horizontal parallelism.

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 need to be found. An example of this perspective can be found in [24], where process communication and plan sharing and merging are discussed. Our proposed approach could be viewed as a special case of system in view of [24], where one process distributes the work and all processes search for a plan until one is found.

Observe that the planning problems we are dealing with in this work are significantly different from the type of planning encountered in other domains, e.g., motion planning (as used for robots). In these situations, the planning involves moving across a static (though possibly not completely known) graph using relatively uniform moves. Often there is a single type of move, possibly parameterized by a simple cost information. Planning is mostly reactive/on-line, while in our framework we are dealing with off-line planning, and typically dealing with a lower dimensionality space but with the added complexity of imprecise actions. Parallel and distributed algorithms for motion planning have been studied (e.g., [70, 2]), typically based on a hierarchical decomposition of the state space and the use of separate threads to develop a motion plan within each region. The work in [70] addresses the additional problem of coordination between multiple agents cooperating in solving the path-finding problem.

Some of the design decisions adopted in this work have their roots in the existing literature on parallel execution of logic and constraint programming [54, 31, 15, 60]. In particular, the perspective of segmenting a search tree horizontally and vertically corresponds to the two traditional forms of parallelism recognized in logic programming (*and-parallelism* and *or-parallelism*). Nevertheless, the actual techniques used in both or-parallelism and and-parallelism are, for the most part, not directly applicable to our context. Execution models for or-parallelism do not account for a heuristic search, and they include further limitations dictated by the desire of reproducing the familiar semantics of logic programming languages [56] (i.e., depth-first left-to-right exploration of the search tree). Similar considerations hold for and-parallelism, which is deeply different from the what we consider here as horizontal parallelism. The former might span segments of branches intermixed by choice points, while horizontal parallelism focuses on an individual edge of the search tree.

Another domain where similar investigations have been performed is operation research. Several investigations have studied the parallelization of branch-and-bound algorithms to tackle hard combinatorial problem. In fact, branch-and-bound algorithms offer similar strategies for exploitation of parallelism—e.g., Crainic et al. [20] recognize two parallelization strategies:

- *node-based* strategies (referred to as parallelism of type 1 in [27])—effectively corresponding to horizontal parallelism, and
- *tree-based* strategies (referred to as parallelism of type 2 in [27])—corresponding to vertical parallelism.

The large majority of the proposed systems deals with tree-based strategies [27, 48, 20].

Vertical parallelism is also related to the literature on parallelization of other search-based procedures. The literature is rich (e.g., [53, 41, 59, 18, 54, 57]), yet such solutions have to be

still customized to the structures of specific problem at hand. Let us also point out that our goal is not to propose a generic algorithm for parallel search, as our experience in several other domains (e.g., logic programming, constraint programming, satisfiability solving) shows that generic parallel search algorithms do not provide the desired level of scalability and efficiency.

7. CONCLUSION 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 planners on multi-core and distributed platforms. Although the use of parallelism in reasoning about actions and change has been explored in the literature, to the best of our knowledge this paper presents the first uniform characterizations of the distinct types of parallelism in heuristic search planning and experimental comparison between them. We identified two major forms of parallelism and investigated their exploitation. The results are very encouraging, in terms of improved execution time, speedups, and scalability.

Although the techniques have been explored in the context of heuristic search planners, they are general and we believe they can be applied to other classes of planning systems. In related publications, we have explored the positive impact of vertical parallelism on planners based on logic programming solvers [3, 44].

We recognize that this is the first step of a long-term research vision. In particular,

- The focus of our effort has been placed on specific classes of planning systems; we believe such classes are important and have played an important role in practical deployments of planning systems.
- Our exclusive focus in the handling of parallelism has been on the search for improved performance and execution time; there are other aspects of planning that can benefit from parallelism, in particular in improving the quality of plans (e.g., length, cost).

A wide array of options have been opened by this research and deserve further exploration. Some of the main avenue of research we intend to pursue are the following:

1. *Implementations on Multi-core Platforms:* the shared memory implementations reported in this paper provide good results, yet they are experimental in nature and can be significantly optimized. The existing literature on parallel execution of logic and constraint programming (e.g., [54, 31, 15]) provides several alternative ways for optimizing the implementation of parallel search systems. Several of such techniques have the potential of improving the implementation of vertical parallelism in planning. In particular, we believe that significant improvements could be achieved by moving from an *asymmetric scheduling* structure, where dedicated processors are used to exclusively handle scheduling operations—as is the case of our approach, with a centralized

queue—to a *symmetric scheduling* organization. In symmetric scheduling, each processor alternates between concrete computation (i.e., search for plans) and scheduling (i.e., balancing of load with other processors). In particular, this implies that a centralized queue is replaced by several local queues, which are balanced through explicit load balancing operations. Similar solutions have been successfully adopted to parallelize other search-based problems (e.g., as done in [43] for answer set programming systems). Symmetric scheduling with local queues has the potential to reduce the contention on the shared resource and enhance scalability. On the other hand, this requires a complete redesign of the planning agents, to enable dynamic load balancing and alternation between computation and scheduling. Furthermore, it requires the investigation of customized strategies for scheduling (e.g., as discussed in [44]) and the introduction of some form of granularity control (to avoid ineffective load balancing operations). All these aspects require extensive research and empirical investigation.

Another important aspect to be explored relates to the peculiar characteristics of modern multi-core architectures. Believing that a single technique, perhaps drawn from the past literature, can address the use of multi-core platforms, is misleading. Multi-core architectures are complex, with memory performance far more critical that old fashioned shared memory platforms. For example, [60] reports a simple experiment, where separate threads running on different cores access random positions of the same very large object; performance, that in the ideal case should be independent of the number of cores, instead rapidly decreases with increased number of threads—due to conflicts on cache access. Extracting parallelism, especially at a fine grained level, requires understanding memory access patterns, and adapting scheduling mechanisms accordingly.

2. *Implementations on Beowulf Clusters:* the distributed memory implementations are also experimental. The prototypes have been effective in confirming the potential scalability on larger platforms. The performance of these implementations can be greatly enhanced. The use of a master-slave structure in the distributed vertical implementation is expected to be a bottleneck for planning problems with large search spaces and fine grained tasks (e.g., alternatives leading to fast failures). This has been observed in similar parallel problems (e.g., [44]). Schemes for fully distributed dynamic scheduling (e.g., based on the stack-splitting principles [56]) have been proposed and should be evaluated in this context.
3. *Evaluation and Optimization:* along with the investigation of schemes to enhance performance and scalability, it is important to investigate other performance parameters. Memory consumption is a factor that could be critical to the performance of a parallel implementation—especially in consideration of the significant growth in the number of states explored by the parallel system w.r.t. a sequential planner. In our experiments, we did not encounter any challenging situations in terms of memory consumption, but a formal study in this direction is needed.

   Another factor is the *quality* of the plans produced by the parallel planners. We have repeatedly observed in our experiments relevant improvements in the quality of the plans produced by the parallel planners with respect to the plans returned by the corresponding sequential implementations. This is particularly true for the case of vertical

parallelism, where alternative search paths are concurrently explored. Nevertheless, a formal investigation of the impact of parallelism on plans quality is missing.

4. *Other Planning Structures:* the design we described in this work focused on progression-based heuristic search planners. We conjecture that similar parallel structures can be effective in other classes of planning systems. For example, several regression search planners (e.g., HSP_r [8]) organize their search according to patterns very similar to those described in Figure 1, and thus can provide analogous forms of parallelism and similar benefits.

   Other planning solutions are expected, instead, to provide significantly different opportunities for exploitation of parallelism. For example, hierarchical task planning (HTN [26]) creates the opportunity for a divide-and-conquer style of parallelization (see, e.g., [69])—where tasks are concurrently decomposed by different processors. These alternative schemes are interesting and should be properly explored.

Acknowledgments

The authors wish to thank the anonymous referees for their insightful comments, that greatly enhanced the content of the manuscript. The research has been supported by NSF grants 0420407, 0812267 and 0220590.

REFERENCES

1. F. Bacchus. The AIPS'00 Planning Competition. *AI Magazine*, 22(3), 2001.
2. S. Balakirsky and O. Herzog. Parallel Planning in Partitioned Problem Spaces. *Symposium on Intelligent Autonomous Vehicles*, `www.isd.mel.nist.gov/documents/publist.htm`, Elsevier, Lisbon, 2004.
3. M. Balduccini, E. Pontelli, O. Elkhatib, H. Le. Issues in Parallel Execution of Non-monotonic Reasoning Systems. *Parallel Computing*, 31:608–647, 2005.
4. C. Baral and M. Gelfond. Reasoning Agents in Dynamic Domains. *Logic Based Artificial Intelligence*, pages 257–279. Kluwer Academic Publishers, 2000.
5. C. Baral, V. Kreinovich, R. Trejo. Computational Complexity of Planning and Approximate Planning in the Presence of Incompleteness. *Artificial Intelligence Journal*, 122(1–2):241–267, 2000.
6. P. Bertoli, A. Cimatti, M. Roveri, P. Traverso. Planning in Nondeterministic Domains under Partial Observability via Symbolic Model Checking. *International Joint Conference on Artificial Intelligence*, pages 473–478, AAAI Press, 2001.
7. A. Blum and M. Furst. Fast Planning through Planning Graph Analysis. *Artificial Intelligence Journal*, 90:281–300, 1997.
8. B. Bonet and H. Geffner. Planning as Heuristic Search: New Results. *European Conference on Planning*, pages 360–372, Springer Verlag, 1999.
9. B. Bonet and H. Geffner. Planning as Heuristic Search. *Artificial Intelligence Journal*, 129(1–2):5–33, 2001.
10. B. Bonet and B. Givan. Results of Conformant Track in the 5th International Planning Competition. `www.ldc.usb.ve/~bonet/ipc5/docs/results-conformant.pdf`, 2006.
11. R. Brafman and J. Hoffmann. Conformant Planning via Heuristic Forward Search: A New Approach. *Artificial Intelligence*, 170(6–7):507–541, 2006.
12. D. Bryce, S. Kambhampati, D.E. Smith. Planning Graph Heuristics for Belief Space Search. *Journal of AI Research*, 26:35–99, 2006.
13. C. Castellini, E. Giunchiglia, A. Tacchella. SAT-based Planning in Complex Domains: Concurrency, Constraints, and Nondeterminism *Artificial Intelligence*, 147:85–117, July 2003.
14. D. Chapman. Planning for Conjunctive Goals. *Artificial Intelligence Journal*, 32(3):333–377, 1987.

15. J. Chassin de Kergommeaux and P. Codognet. Parallel Logic Programming Systems. *ACM Computing Surveys*, 26(3):295–336, 1994.
16. A. Cimatti, M. Roveri, P. Traverso. Strong Planning in Non-deterministic Domains via Model Checking. *Artificial Intelligence Planning Systems*, pages 36–43, AAAI/MIT Press, 1998.
17. A. Cimatti et al. Conformant Planning via Symbolic Model Checking and Heuristic Search. *Artificial Intelligence Journal*, 159:127–206, 2004.
18. D. Cook and C. Hannon. Adaptive Parallel Search for Theorem Proving. *FLAIRS Conference*, pages 351–355, 1999.
19. D. Corkill. Hierarchical Planning in a Distributed Environment. *International Joint Conference on Artificial Intelligence*, pp. 168–175, 1979.
20. T.G. Crainic, B. Le Cun, and C. Roucairol. Parallel Branch and Bound Algorithms. In E.G. Talbi editor, *Parallel Combinatorial Optimization*, John Wiley & Sons, pp. 1–28, 2006.
21. M. desJardins, M. Wolverton. Coordinating Planning Activity and Information Flow in a Distributed Planning System. *AAAI Fall Symposium on Distributed Continual Planning*, 1998.
22. Y. Dimopoulos, A. Gerevini, P. Haslum, A. Saetti. The Benchmark Domains of the Deterministic Part of IPC-5. *Abstract Booklet of the competing planners of the Fifth International Planning Competition - Satellite Event of the Sixteenth International Conference on Automated Planning & Scheduling*, pp. 14–19, 2006.
23. Y. Dimopoulos, A. Gerevini, A. Saetti. The Pathways Domain. `zeus.ing.unibs.it/ipc-5/domain-descriptions/pathways.txt`.
24. E. H. Durfee. Distributed Problem Solving and Planning. In *Multiagent Systems*, eds G. Weiss, pp. 121–164. The MIT Press. 1999.
25. T. Eiter, A. Polleres, G. Pfeifer, N. Leone, W. Faber. A Logic Programming Approach to Knowledge State Planning, II. *Artificial Intelligence*, 144(1-2), 2003.
26. K. Erol, D. Nau, and J. Hendler. HTN Planning: Complexity and Expressivity. *National Conference on Artificial Intelligence (AAAI)*, pages 1123–1128, AAAI/MIT Press, 1994.
27. B. Gendron and T.G. Crainic. Parallel Branch-and-Bound Algorithms: Survey and Synthesis. *Operations Research*, 42(6):1042–1066, 1994.
28. A. Gerevini and D. Long. Plan Constraints and Preferences in PDDL3: The Language of the Fifth International Planning Competition. Technical Report, University of Brescia, Italy, 2005.
29. A. Gerevini, B. Bonet, B. Givan. Fifth International Planning Competition. `www.plg.inf.uc3m.es/icaps06/preprints/i06-ipc-allpapers.pdf`, 2006.
30. A. Grama and V. Kumar. State of the Art in Parallel Search Techniques for Discrete Optimization Problems. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):28–35, 1999.
31. G. Gupta, E. Pontelli, M. Hermenegildo, M. Carlsson, K. Ali. Parallel Execution of Prolog Programs: a Survey. *ACM Transactions on Programming Languages and Systems*, 23(4):472–602, 2001.
32. P. Haslum, B. Bonet, H. Geffner. New Admissible Heuristics for Domain-independent Planning. *National Conference on Artificial Intelligence (AAAI)*, AAAI/MIT Press, pp. 1163–1168, 2005.
33. J. Hoffmann and B. Nebel. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.
34. J. Hoffmann and S. Edelkamp. IPC-4 Home Page. `ls5-web.cs.uni-dortmund.de/~edelkamp/ipc-4/domains.html`.
35. M. Iwen and A. Mali. Distributed Graphplan. *International Conference on Tools with Artificial Intelligence*, pp. 138, IEEE Computer Society, 2002.
36. J. Jaja. *An Introduction to Parallel Algorithms*. Addison Wesley, 1992.
37. F. Kabanza, L. Shuyun, S. Goodwin. Distributed Hierarchical Task Planning on a Network of Clusters. *Int. Conference on Parallel and Distributed Computing and Systems*, ACTA Press, pp. 439–444, 2004.
38. H. Kautz and B. Selman. Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search. *National Conference on Artificial Intelligence (AAAI)*, AAAI/MIT Press, pp. 1194–1199, 1996.
39. H. Kitan and J.A. Hendler. *Massive Parallel Artificial Intelligence.* MIT Press, 1994.
40. J. Koehler and K. Schuster. Elevator Control as a Planning Problem. *Artificial Intelligence Planning Systems*, pp. 331–338, AAAI Press, 2000.
41. R. Korf and P. Schulte. Large-scale Parallel Breadth-First Search. *National Conference on Artificial Intelligence (AAAI)*, pages 1380–1385, AAAI/MIT Press, 2005.
42. H. Le. Efficient Parallel Execution of Answer Set Programs. Ph.D. Dissertation, New Mexico State University, Department of Computer Science, 2007.
43. H. Le and E. Pontelli. An Investigation of Sharing Strategies for Answer Set Solvers and SAT Solvers. *EuroPar*, Springer Verlag, pp. 750–760, 2005.

44. H. Le and E. Pontelli. Dynamic Scheduling in Parallel Answer Set Programming Solvers. *High Performance Computing Symposium*, ACM Press, 2007.
45. H-F. Leung. *Distributed Constraint Logic Programming.* World Scientific Pubs. 1993.
46. V. Lifschitz. Answer set programming and plan generation. *Artificial Intelligence Journal*, 138(1–2), 2002.
47. A.D. Mali and S. Kambhampati. Distributed Planning. In Urban, J., and Dasgupta, P. (eds), *Encyclopedia of Distributed Computing*, Kluwer, 2003.
48. B. Mans and C. Roucairol. Performance of Parallel Branch-and-Bound Algorithms with Best-First Search. *Discrete Applied Mathematics*, 66(1):57–74, 1996.
49. D.V. McDermott et al. PDDL: The Planning Domain Definition Language. Technical Report, Yale University, August 1997. `ftp.cs.yale.edu/pub/mcdermott/software/pddl.tar.gz`
50. D.V. McDermott. The 1998 AI Planning Systems Competition. *AI Magazine*, 21(2):35–55, 2000.
51. L. Michel, A. See, P. Van Hentenryck. Parallelizing Constraint Programs Transparently. *Principles and Practice of Constraint Programming*, Springer Verlag, pp. 514–528, 2007.
52. P. Moura, R. Rocha, and S.C. Madeira. Thread-based Competitive Or-Parallelism. *International Conference on Logic Programming*, Springer Verlag.
53. J. Oplinger, D. Heine, M. Lam. In Search of Speculative Thread-level Parallelism. *Parallel Architectures and Compilation Techniques*, pages 303–313, IEEE Computer Society, 1999.xs
54. L. Perron. Search Procedures and Parallelism in Constraint Programming. *Principles and Practice of Constraint Programming*, Springer Verlag, pp. 346–360, 1999.
55. M.C. Pinotti and G. Pucci. Parallel Algorithms for Priority Queue Operations. *Theoretical Computer Science*, 148(1):171–180, 1995.
56. E. Pontelli, K. Villaverde, H. Guo, G. Gupta. Stack Splitting: a Technique for Efficient Exploitation of Search Parallelism on Share-Nothing Platforms. *Journal of Parallel and Distributed Computing*, 66(10):1267–1293, 2006.
57. T.K. Ralphs, L. Ldanyi, and M.J. Saltzman. A Library Hierarchy for Implementing Scalable Parallel Search Algorithms. *Journal of Supercomputing*, 28(2):215–234, 2004.
58. R. Rönngren and R. Ayani. A Comparative Study of Parallel and Sequential Priority Queue Algorithms. *ACM Transactions on Modeling and Computer Simulations*, 7(2):157–209, 1997.
59. N. Prcovic. Changing the Distribution Depth During a Parallel Search Tree. *EuroPar*, pages 1217–1220, Springer Verlag, 1997.
60. V. Santos Costa. On Supporting Parallelism in a Logic Programming System. *Declarative Aspects of Multicore Programming*, `www.clip.dia.fi.upm.es/Conferences/DAMP08`, 2008.
61. D. Smith and D. Weld. Conformant Graphplan. *National Conference on Artificial Intelligence (AAAI)*, pages 889–896, AAAI/MIT Press, 1998.
62. T. C. Son, P.H. Tu, M. Gelfond, R. Morales. Conformant Planning for Domains with Constraints. *National Conference on Artificial Intelligence (AAAI)*, pages 1211–1216, AAAI/MIT Press, 2005.
63. G. Stolting Brodal. Priority Queues on Parallel Machines. *Parallel Computing* 25(8):987–1011, 1999.
64. S. Thiebaux, J. Hoffmann, and B. Nebel. In Defense of PDDL Axioms. *Artificial Intelligence*, 168:(1-2), pages 38–69, 2005.
65. H. Turner. Polynomial-Length Planning Spans the Polynomial Hierarchy. European Conference on Logics in Artificial Intelligence (JELIA), pages 111–124, Springer Verlag, Lecture Notes in Computer Science 2424, 2002.
66. J. Urban, P. Dasgupta. *Encyclopedia of Distributed Computing.* Kluwer, 2003.
67. K. Villaverde and E. Pontelli. An Investigation of Scheduling in Distributed Constraint Logic Programming. *International Conference on Parallel and Distributed Computing Systems*, pages 98–103, ISCA, 2004.
68. D. Vrakas, I. Refanidis, and I. P. Vlahavas. Parallel planning via the distribution of operators. *Journal of Experimental and Theoretical AI*, 13(3):211–226, 2001.
69. B. Wilkinson and M. Allen. *Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers.* Prentice Hall, 2004.
70. M. Yokoo and T. Ishida. Search Algorithms for Agents. *Multiagent Systems*, pages 165–199, MIT Press. 1999.
71. H. Zhang, M.P. Bonacina, J. Hsiang. PSATO: a Distributed Propositional Prover and its Application to Quasigroup Problems. *Journal of Symbolic Computing*, 21(4), 1996.