DISTRIBUTED HIERARCHICAL TASK PLANNING ON A NETWORK OF CLUSTERS

Froduald Kabanza
Computer Science
University of Sherbrooke
Sherbrooke, Qc J1K 2R1 Canada
kabanza@usherbrooke.ca

Shuyun Lu
Computer Science
University of Windsor
Windsor, ON N9B 3P4 Canada
lu11@uwindsor.ca

Scott Goodwin
Computer Science
University of Windsor
Windsor, ON N9B 3P4 Canada
sgoodwin@uwindsor.ca

ABSTRACT
A wide range of planning applications are combinatorial in nature, making the design of general purpose planning algorithms a still very challenging endeavor. In order to cope with this combinatorial complexity, some of the most recent work in artificial intelligence (AI) planning focuses on the use of sophisticated heuristics, domain search control knowledge, random search and efficient abstract state space encodings such as binary decision diagrams. The additional performance needed by complex planning applications can be provided by adopting massively parallel computing systems, such as networks of clusters. This paper describes a simple, general approach for turning backtrack search based planners into more powerful distributed systems that run on networks of clusters. Our approach consists in distributing backtrack search points to different processes on the network. We illustrate its potential using DSHOP, a distributed version of the SHOP planner.

KEY WORDS
Task planning, distributed planning, distributed search

1. Introduction

Planning is an important activity in many control and decision making problems, such as robot navigation, and robot task planning, speech generation, and even computer animation. There exist many different approaches to planning, depending on the representations of goals and actions, the system and environment models, and in the end the actual algorithms used to reason about those representations and models to generate plans [1].

A common feature of most planning techniques is the explicit exploration of some state space. This is also their main source of limitations, since state spaces tend to grow too large, making automated planning hopeless for many applications. In order to cope with the state explosion problem, a significant part of recent research in AI planning has focused on efficient heuristic search techniques [2], efficient search space representations and corresponding planning algorithms [3], knowledge-based search-control [4], random search [5] and abstract state representations such as binary decision diagrams [6].

The use of massively parallel computing systems (also known as high-performance computing systems) can provide the additional level of performance needed by these planning systems. Nowadays high-performance systems are marketed by all the major computer vendors, have entered the main-stream market, and are freely available in most academic institutions. Considering that these systems can comprise hundreds and sometimes thousands of processors, this creates opportunities for exploring high-performance planning systems which, as would be expected, can solve much more complex planning problems.

We present an approach for reformulating backtrack search based planners into more efficient distributed versions that run on network of clusters, a high-performance computing model made of workstations or PCs interconnected through a LAN. The key idea behind our approach is to distribute backtrack choice points in the exploration of a search space to different processes on a network of clusters. The presence of nondeterministic choices in the search process is enough to make a difference. Since backtrack choices in a planning system can result from a model of concurrency by interleaving, obviously problems involving concurrent actions will benefit from our approach. But we also have backtrack choices resulting from the selection of actions at different points in the search process, hence any planning problem can benefit from our approach even without involving any concurrent actions at all.

SHOP is a deterministic planning system that generates plans by searching through a space of task decompositions; it is very expressive and if carefully implemented, it is one of most efficient deterministic planning approaches [6]. We developed a distributed version of SHOP that we call DSHOP, based on a Java version SHOP [7,8]. Preliminary results show that, even naïvely implemented (without any heuristics, search-control, or optimized data structures), DSHOP yields considerable savings.

The remainder of the paper is organized as follows. The next section explains the cluster computing model. The following section discusses the basic principles behind our distributed planning approach. Then, we explain how these principles are applied to design DSHOP algorithm. This is followed with a presentation on some preliminary results, then a discussion on related works, and finally a brief concluding note.

2. Cluster Computing

In the cluster computing model, every processor executes its own copy of the same program, possibly on different data. There is no shared memory among different processors. In other words, this is a *single program, multiple data* (*SPMD*) model, which is a particular case of the classic *single instruction, multiple data* (SIMD). Processors are grouped into clusters, which are interconnected to form a network of clusters, all this in a manner transparent to the user. Each cluster is also called a *node* (that is, a node of computation). A program running on a processor is also called a *worker*. To turn a network of clusters into a virtual parallel computer, one just needs *message passing* software that supports inter-process communication.

3. Simple Distributed Backtrack Search

Like many planning systems, SHOP relies on backtrack search. Before giving a detailed algorithm for DSHOP, it is useful to outline the basic principles behind it in the more general terms of backtrack search. Once one understands those principles and the original SHOP algorithm, it becomes easier to understand DSHOP algorithm.

SHOP planning algorithm can be seen as exploring an or-tree, in which each or-link models nondeterministic choices (or, equivalently a backtrack points). The objective of the exploration is to find a satisfactory path, that is, a plan. For now, it does not matter how nodes in the tree are encoded and how successor nodes are computed. What is important is to know that we have an initial node from which the exploration begins, we have a mechanism of determining successors of the current node, and we have a mechanism of telling whether the current node is a success, a failure or none of those.

Thus, once we abstract away the representation of nodes in the search tree, the computation of successors and the determination of successful or failing nodes, SHOP algorithm, like many other planning systems, becomes basically a standard backtrack search algorithm.

3.1 Backtrack Search on a Single Processor

On a single processor, backtrack search proceeds as follows. It starts from an initial node and expands it; if this yields more than one successor, it picks one them and expands it; this process of picking a node and expanding it continues recursively until either we reach a successful node or a failure node; on a successful node, we simply report the current path as a solution; on a failure node, we backtrack to the most recent node having an explored successor and resume the recursive node expansions from there (intelligent backtracking heuristics can be involved at this stage to pick, not the most recent one, but one that is deemed more promising).

3.2 Backtrack Search on Distributed Processors

For a distributed version, whenever a node has more than one successor, one of them (let’s say the first) is explored by the current worker, and for each of the others we spawn a different worker.

More precisely, we elect one worker to act as *a manager*, which will tell other workers from which search node to begin their recursive state expansions. When a worker receives a node from the manager, it begins searching from this node, by repeatedly expanding nodes as in the single processor case, except for what follows:

- whenever the current search path fails, the worker does not backtrack, instead it becomes free, so that the manager can assign it another node from which to start;
- whenever the worker encounters a backtrack point (i.e., a node with more than one successor), it sends its successor nodes to the manager (except one it will pursue searching from), so that it knows about them and assigns them workers;

We can have variations of the above procedure, by letting workers do a controlled number of backtrackings before sending any successor nodes to the manager.

Since we do not have shared memory in the cluster computing model, the only possible way for a manager to tell a worker the start node and for a worker to inform the manager about encountered nondeterministic choice points is by exchanging messages. We distinguish between *node copy* and *node recomputation* modes for the exchange of nodes between the manager and the workers.

3.3 Node Copying

In a node copying mode, the whole node is exchanged (i.e., the manager sends a complete node to the worker and the worker sends a list of unexplored successor nodes to the manager). The implementation of this mode is straightforward; all that is needed is to send a message containing all the relevant node information.

3.4 Node Recomputation Mode Using Oracles

Sending and receiving nodes between the manager and the workers can slow down the performance, because in DSHOP a node contains a large amount of data. An alternative is node recomputation, which consists in passing an *oracle* to the node, that is, a sequence of integers guiding a state space exploration down to a node without any backtracking. We borrow this idea of using oracles from a parallel version of Prolog described by [9].

Given an oracle {i, j, …, k}, a worker reaches the corresponding node by taking the *(i-1)* successor on the first backtrack point (this is not necessarily the first node; it's the first node having more than one choice), then the *(j-1)* successor on the second backtrack point*, and* so on until the last choice point. Figure 2 gives an illustration. The oracle to node S1 is {0,1} and the oracle to node S2 is {1,2}. Given oracle {1,2}, a worker takes the second successor on the first backtrack point, then the third successor of the second backtrack choice point, which leads to S2.

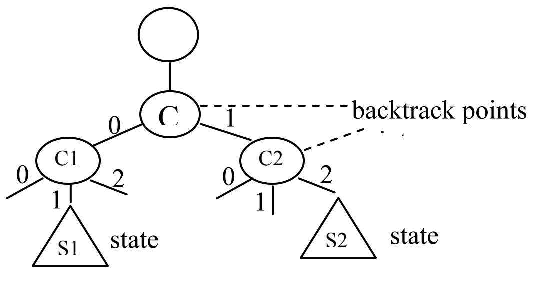

Figure 1: oracles

Upon receiving an oracle from the manager, the worker follows it, that is, it deterministically takes the indicated choices, without backtracking, down to the end of the oracle; the worker resumes his normal execution from the reached node, but this time it must record oracles for encountered backtrack points and send to the manager in place of nodes: the initial node has an empty oracle; if the current node has just one successor, the successor inherits the parent node's oracle, but no oracle is sent to the manager; otherwise, if the current node has more than one successor (let's say assume *m* successors), the oracle of the *i*-th successor (the order does not matter) is adding *i* at the end of the parent's oracle; hence an oracle is sent only when it is updated.

It's impossible to tell *a priori* which mode is more efficient between state copy and state recomputation. On the one hand, since an oracle is more compact than a node, it incurs less communication overhead. On the other hand, following an oracle requires recomputing a path; this may turn out to be expensive depending on how the computation of successor nodes is implemented. SHOP computation of successor nodes involves a pattern matching between method templates and the current node. In the long run, the cost of pattern matching may outweigh the reduction in communication overhead, particularly if the pattern matching algorithm is naively implemented.

It can be demonstrated that a distributed backtrack search algorithm implemented following the above guidelines preserves the coverage of the search space of the original single-processor backtrack search algorithm, however, a formal proof is beyond the limits of this paper.

4. Distributed SHOP

DSHOP is based upon JSHOP, a Java implementation of SHOP [7,8]. DSHOP uses exactly the same language for representing domain applications and problems as JSHOP. A *domain* application is specified by describing *methods* that decompose tasks in this domain into subtasks, and *horn-clauses* that define derived predicates involved in the method definitions. Tasks are classified into non-primitive tasks and primitive ones. Methods are in turn classified into primitive methods (called *operators* henceforth) and non-primitive methods (simply called *methods* henceforth). Operators are actually action templates and model primitive actions like operators do in most planning systems. Given a domain definition, a *planning problem* is specified by giving an *initial world state* and a *list of tasks*. Solving the problem means to find a sequence of decompositions of all the initial tasks down to primitive tasks. The solution is the resulting sequence of primitive tasks. Note that there is no explicit goal specification as input; this is implicitly conveyed by the modeling of tasks.

As in JSHOP, a *node* in DSHOP consists of a world state, a list of tasks and a current partial plan (which is a sequence of primitive operators). The initial node is formed of the input world state, the input task list and an empty plan. The domain is a global variable left understood in the algorithms below. Because of space limitations, below we only explain the state-copy mode of DSHOP, abstracting away the oracle bookkeeping for state-recomputation and the selection of an execution mode. The manager's algorithm is as follows.

```
Procedure Manager()
send an empty oracle to one of the workers
do  // loop
  M = get a message from communication buffer
  if M is a solution then
     send stop message to every worker
     return solution
  endif
  if M is a list of nodes then add them to
     list of backtrack nodes
  endif
  while there is backtrack node N and idle worker W
        send N to W;  delete N from backtrack nodes
  enddo
  if there is no job and all workers are idle then
    return FAIL // No solution can be found
  endif
enddo
end Manager
```

The worker's algorithm proceeds as follows. It invokes DSHOP, which performs the actual search and is described further below.

```
Procedure Worker()
do
  M = received message from manager
  if M is a node N then
    r = DSHOP(N)
    send r to manager
  else if M is the stop message then break endif
enddo
end Worker
```

DSHOP algorithm recursively expands the current node into successors. If the top task is primitive (Line 6), the expansion of a node consists in finding all operators matching this task; for each of them, *op*, we get a new node *(op(S),T, add(op, P))* (Line 7), where *op(S)* is the state obtained applying operator *op t*o *S*, and *add(op, P)* is the plan obtained by adding *op* in front of plan *P*; one of the node is expanded recursively (Line 11); all the others are sent as backtrack nodes to the manager (Line 8-9).

```
Procedure DSHOP(N )
1  T= N.task; P = N.plan; S = N.State
2  if N.tasks = nil then return nil
   endif
     t = the first task in N.tasks
    T = the remaining tasks in N.taks
6   if t is primitive and there is an operator for t then
7     M = list of tuples (op(S), T,add(op,P))
            for each operator for t
8     if length(M) > 1 then
9          Send the rest(M) to manager // backtrack points
      endif
11    return DSHOP(S, T, P)
    endif
13  if t is non-primitive and there is a
        simple reduction of t in S then
       M = list of tuples (S,append(R,T),P)
            for each reduction of t;
15     if length(M) > 1 then
          send rest(M) to manager // backtrack points
       endif
16     return DSHOP(first(M))
       endif
17  else return FAIL // No plan
    endif
end DSHOPC
```

If the top task is not primitive (Line 13), the expansion of a node consists in finding all methods that match the top task in order to reduce it into a list of smaller tasks *R*; each matching method gives one set *R*, then for each *R* we get a new node *(S, append(R,T), P)* (Line 7), where *append(R,T)* is a list obtained by concatenating *R* and *T*. If no operator or method matches the top task, then DSHOP fails (i.e., the process will become free).

It can be shown that DSHOP sooner or later terminates with a correct solution if there exist any. In other words, DSHOP is sound and complete, just as is SHOP. A formal proof is, however, out of the limits of this paper.

4. Experiments

The DSHOP planning system is implemented using Message Passing Interface (MPI), which is a library specification for message passing, proposed as a standard by a broadly based committee of vendors and public institutions [10]. MPI include many primitive for inter-communication process; the following are sufficient for our implementation: *initializing MPI* (invoked prior to any other MPI call in order to set up the MPI environment), *closing up MPI* (invoked at the end of an application); *requiring processors* (allocates processors for an application); *identifying processors* (identifies the current processor); *sending a message* (to another worker); and *receiving a message* (from another work).

We made preliminary experiments of a DSHOP implementation on a network of eleven geographically distributed high-performance computing clusters. We used a node that has one 12 and one 36 four-processor, 833Mhz, Alpha SMP (symmetric multi-processors) cluster connected via Quadrics interconnection technology.

We performed many experiments on a set of randomly generated artificial domains with a set amount of backtracking and a few experiments on the blocksworld domain. The problems on the artificial domains were generated with a fixed average branching factor and a single solution placed at the rightmost node (worst case). In both domains, we collected the elapsed time, exchanged message size, recomputing time, actual working time, idle time, and data format converting time for each test run. For DSHOP, the elapsed time is calculated from when the manager sends out the first message to a worker until it receives the first message with a solution from one of the workers. For JSHOP, the elapsed time is calculated from when the planner starts searching until finds a solution.

Of the many experiments we carried out, the most significant results can be summarized in the following two performance plots. Figure 2 shows that as the average branching factor is increased the runtime of JSHOP (with one processor) increases as expected. For both DSHOPC and DSHOPR, the runtime grows more slowly indicating parallelism is being exploited. DSHOPC outperforms DSHOPR which indicates the state size is small relative to the complexity of recomputation in our problem set.

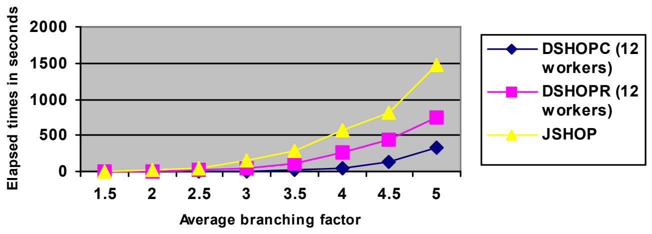

Figure 2: DSHOPC vs. DSHOPR vs. JSHOP

For both DSHOPC and DSHOPR, at every backtrack point, all choices but one are sent to the manager. When a worker finishes a branch, it is often reassigned a choice it gave away earlier. This is clearly wasteful. What is needed is a load balancing strategy which encourages workers to keep choices when there are no idle processors. We intend to pursue this in the future. But first, we show the potential gains of limiting the choices given away. One simple policy is to only give away choices in the top portion of the tree. DSHOPC-5 and DSHOPR-5 keep all choices below level 5 in the search tree. In our experiments, the performance of DSHOPC-5 and DSHOPR-5 are pretty close. Figure 3 shows the results when the average branching factor is 4.5. In our future work, there are numerous facets we wish to investigate such as the effect of state size and/or recomputation complexity versus performance, and using hundreds of processors on very large problems.

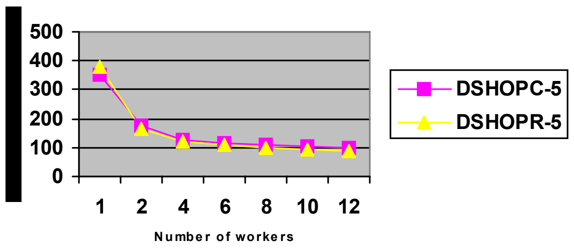

Figure 3: DSHOPC vs. DSHOPR

For rapid prototyping the preliminary DSHOP implementation does not include efficient data structures (e.g., states are simply stored as lists and the unification process does not remember previously matched patterns). Also, DSHOP runs recursively almost as described above, rather than using explicit iterative loops. With a more efficient implementation, both DSHOPC and DSHOPR will obviously run faster, and we do expect DSHOPR to outperform DSHOPC on additional problems since the overhead in recomputing a path will become reduced by a more efficient pattern matching.

4. Related Work

Most of the work on distributed planning focuses on decomposing goals or problems and distributing them to multiple agents [11, 12,13]; Corkill's distributed version of NOAH planner [14] is on of the oldest examples of this. More recently, desJardins and Wolverton developed DSIPE, a distributed version of SIPE [15]. In DSIPE, a manager process partitions sub-goals among workers; then each worker tries to find a subplan for the subgoal; finally the manager merges them. DGP is another distributed plan that is also based on goal-decomposition, but based on GRAPHPLAN [16]. DSHOP distributes complete computation paths rather than partial computation paths (i.e., computations of partial plans corresponding to subgoals) as is the case in DSIPE and DGP. In fact, we think that both ideas can be integrated to yield an even more effective distributed planning model; we intend to investigate this in the future.

The idea of decomposing a planning problem into smaller problems that are solved by distributed agents has also been examined in the context of nondeterministic planning problems. In particular, for Markov decision planning (MDP) problems, Guestrin and Gordon described an algorithm that represents an MDP problem as a linear programming problem, represented using a compact, factored representation, and solved by decomposing it and then assigning its parts to planners distributed on a network, using message passing to communicate among agents [17]. They did not report on any implementation of this algorithm, but clearly it's a good candidate for experimentation on a network of clusters. The main difference with our approach is that, like the other approaches above, this technique is based on goal-decomposition; more precisely, since goals in MDP planning are implicitly conveyed by reward functions, they decompose a Markov decision process into smaller ones, for which distributed agents compute policies, that are then combined to obtain a global policy. In contrast, our approach decomposes the solving procedure based on its nondeterministic choices. A possible direct extension of our approach to the MDP paradigm is first to deal with and-or search spaces and then adapt backtrack search techniques for solving MDP problems such as LAO* [18]. This approach would be complementary to the one by Guestrin and Gordon.

Other planning systems involve problem decompositions that could easily lead to distributed planning even if they are not explicitly designed for this purpose. An example is Lansky's Collage COLLAGE system which uses a technique (called localization) of decomposing a planning problem into smaller problems (called regions) [19]. This technique may be exploited in a high-performance planning environment by having different workers solve problems in different regions. Another example is A-SHOP [20], an agent-based version of A-SHOP. An A-SHOP agent can gather information from distributed sources and communicate with other A-SHOP agents, however there are non multiple copies of A-SHOP running concurrently to solve the same problem.

Knowledge-based distributed problem solvers are not just limited to planning systems. Other examples include for instance distributed logic programming systems, including Prolog [9,21,22].

4. Conclusion

The increasing popularization of high-performance computing, in particular networks of clusters that potentially provide access to thousands of processors, opens up the opportunity to adapt existing planning algorithms in order to solve complex problems that are currently beyond single-processor computing.
The research presented in this paper is a first step in this line of inquiry, which can be pursued in many different directions, including those hinted at in the above discussions on related work and on experiments.
Additional interesting prospects include experimenting with real-world problems and making a web-based interface that would allow users to solve high-performance planning problems on SHARCNET.

4. Acknowledgements

This work is supported by a SHARCNET grant.

References

[1] M. Ghalab, D. Nau & P. Traverso. *Automated Planning: Theory and Practice.* Morgan Kaufmann, 2004.

[2] H. Hoffmann & B. Nebel B. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14:235-302, 2001.

[3] S. Kambhampati, E. Parker & E. Lambrech. Understanding and Extending Graphplan. *Recent Advances in AI Planning*, Lecture Notes in Artificial Intelligence, 1348:260-272, 1997.

[4] F. Bacchus and F. Kabanza. Using Temporal Logic to Express Search Control Knowledge for Planning. *Artificial Intelligence*, 116 (1-2) 123-191, 2000.

[5] H. Kautz & B. Selman. Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search.. In *Proc. of 13th National Conference on Artificial Intelligence*, 1194-1201, 1996.

[6] P. Bertoli, A. Cimatti, M. Roveri & P. Traverso. Planning in Nondeterministic Domains under Partial Observability via Symbolic Model Checking. In *Proc. of 7th Int'l Joint Conference on Artificial Intelligence*, 473-478, 2001.

[7] D. Nau, T. Au, O. Ilghami, U. Kuter, W. Murdock, D. Wu & Y. Yaman. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research* 20:379-404, 2003.

[8] D. Nau, Y. Cao, A. Lotem, & H. Munoz-Avila. The SHOP Planning System. In AI Magazine, 2001.

[9] S. Mudambi & J. Schimpf. Parallel CLP on Heterogeneous Networks. In *Proeedings of the 11th International Conference on Logic Programming.* MIT Press, pp. 124-141. 1994.

[10] G. William, L. Ewing & S. Anthony. Using MPI: *Portable Parallel Programming with the Message Passing Interface.* 2nd Edition. MIT Press, 1999.

[11] E. Durfee. Distributed problem solving and planning, in *Multi-agents systems and applications*. Springer-Verlag, 118-149, 2001.

[12] M. desJardins, E. Durfee, C. Ortiz, & M. Wolverton A Survey of Research in Distributed, Continual Planning. In *AI Magazine 4,* pp. 13-22, Winter 2000.

[13] M. Wolverton & M. desJardins. Controlling communication in distributed planning using irrelevance reasoning. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*. AAAI Press, pp. , 868–874, 1998.

[14] D. Corkill. Hierarchical planning in a distributed environment. In Proceedings of the 6th International Joint Conference on Artificial Intelligence. 168-175, 1979.

[15] M. desJardins & M. Wolverton. Coordinating a Distributed Planning System. In *AI Magazine*, Winter 1999.

[16] M. Iwen & D. Mali. Distributed Graphplan. In 14th *IEEE Conference on Tools with Artificial Intelligence*. 138-145, 2002.

[17] C. Guestrin & G. Gordon. Distributed Planning in Hierarchical Factored MDPs. In *Proceedings of Uncertainty in Artificial Intelligence Conference (UAI)*, 2002.

[18] E. Hansen & S. Zilberstein. LAO*: A heuristic search algorithm that £nds solutions with loops. *Artifi*cial Intelligence, 129:35–62, 2001.

[19] A. Lansky. Localized Planning with Action-Based Constraints. In *Artificial Intelligence* 98(1-2): 49-136.

[20] H. Dix, H. Munoz-Avila, D. Nau & L. Zhang. IMPACTing SHOP: Putting an AI Planner into a Multi-Agent Environment, in: *Annals of Mathematics and AI*, (2000).

[21] J. Schumann& R. Letz. PARTHEO: A high-performance parallel theorem prover. In: *Proc. Of CADE'90,* pp. 40-56, 1990.

[22] E. Shapiro. Or-Parallel Prolog in Flat Concurrent Prolog. In *Journal of Logic Programming,* pp. 243-267, 1989.