Parallel Planning via the Distribution of Operators

Dimitris Vrakas, Ioannis Refanidis and Ioannis Vlahavas
Department of Informatics
Aristotle University of Thessaloniki
[dvrakas, yrefanid,vlahavas]@csd.auth.gr

Abstract

This paper describes ODMP (**O**perator **D**istribution **M**ethod for Parallel **P**lanning), a parallelization method for efficient heuristic planning. The method innovates in that it parallelizes the application of the available operators to the current state and the evaluation of the successor states using the heuristic function. In order to achieve better load balancing and a lift in the scalability of the algorithm, the operator set is initially enlarged, by grounding the first argument of each operator. Additional load balancing is achieved through the reordering of the operator set, based on the expected amount of imposed work. ODMP is effective for heuristic planners, but it can be applied to planners that embody other search strategies as well. It has been applied to GRT, a domain–independent heuristic planner, and CL, a heuristic planner for simple Logistics problems, and has been thoroughly tested on a set of Logistics problems adopted from the AIPS-98 planning competition, giving quite promising results.

1 Introduction

A challenging feature of modern artificial intelligence applications is the ability to distribute the workload among several processors, in order to increase the execution speed. Although the technology of parallel architectures is quite mature and a large number of parallel systems are available at a reasonable cost, there are not many software products that can exploit these capabilities.

Many researchers have tried to find parallelization techniques for AI applications, focusing mainly on ways to distribute the search tree among the existing processors (Rao et al. 1987, Kumar et al. 1988, Powley and Korf 1989, Kumar and Rao 1990, Cook 1997, Cook and Varnell 1999). These techniques, which have been enriched with load balancing (Kumar et al. 1994) and operator reordering (Powley and Korf 1989, Powley et al. 1991, Cook et al. 1993), produce quite efficient parallel algorithms. However, there are two main problems related with the kind of parallelization based on the distribution of the search space: a) a great number of states is examined more than once, since the state-space is not always split in disjoined parts and b) these techniques impose a significant quantity of redundant search in subtrees that do not lead to any solution.

Planners are Artificial Intelligence applications, which given an initial state *I*, a set of actions *A* and a set of goals *G*, produce a sequence of actions (called plan), which if applied to *I* achieves *G*. These programs are in many cases

embedded in systems that must exhibit real-time behavior, so they are usually equipped with heuristic functions and other types of guidance, in order to respond promptly. The heuristic functions speed up the planning process by narrowing down the search in a small subtree around the solution path. Since the size of this subtree is close to the length of the solution, parallelization methods based on the distribution of the search space are ineffective.

This paper describes ODMP (Operator Distribution Method for parallel Planning), an innovative approach for exploiting parallelism in heuristic planning. The approach distributes to the available processors the tasks of a) finding the actions that can be applied to a given state, b) constructing the successor states and c) evaluating the new states using the heuristic function. It also describes three techniques that extend the original method and offer additional speedup in the parallel programs. The extended method has been applied to GRT (Vrakas et al. 1999) and CL (Vrakas et al. 2000) planners with remarkable success.

The rest of the paper is organized as follows: Section 2 reviews related work on the field of parallel planning and search methods. Section 3 presents an overview of ODMP, while section 4 introduces the Operator Reordering Technique, the Semi-Grounded Operators Technique and the Distributed Agenda Architecture Technique, three extensions to the basic parallelization method. Section 5 gives a brief description of the two planning paradigms used in the experiments and section 6 presents the results of the experiments and an analysis on the efficiency of ODMP. Finally, section 7 concludes the paper and poses future directions.

2 Related Work

Kumar et al. (1988) review a set of strategies for parallel best-first search of state-space graphs. The strategies they present are classified to be either distributed or centralized, based on the existence or not of local agendas. In both cases the heuristic function is used to order the states in the agenda, i.e. the first state in the agenda is the one with the smallest estimated distance from a goal state.

In the centralized model, each one of the N processors undertakes the best state of the global agenda, which has not yet been assigned to any other processor. At the end of each expansion the successor states are placed back to the global agenda. The main advantage of this approach, as discussed by Irani and Shih (1986), is that it does not result in much redundant search. However, the global agenda is accessed by all the processors very frequently, causing the processors to remain idle for quite a long time, due to contention.

On the other hand, in the distributed model each processor maintains its own local agenda and thus there is little need for synchronization. This model usually uses the IDA* search algorithm initially presented by Powley et al. (1991). IDA* is a version of Iterative Deepening search, where the next level of search is determined by the heuristic function in use. The state-space is initially divided and

distributed to the existing processors. The segmentation of the initial state-space can be done in several ways. Powley and Korf (1989) introduced PWS, a tree distribution method in which each processor searches at a unique depth.

Kumar, Rao and Ramesh (Kumar and Rao 1990, Rao et al. 1987) describe a different approach where the search tree is segmented vertically, i.e. after a sufficient number of states has been generated, each processor undertakes one of them, considering it to be the root and searches the generated subtree.

A large number of variations of these techniques have been proposed over time. For example, Diane Cook proposed a hybrid approach (Cook and Varnell 1999, Cook 1997), which combines IDA* with vertical segmentation techniques and seems to outperform all the other methods.

In the above methods, after the initial distribution of the state-space, some intercommunication is necessary, since some of the processors may be working on promising parts of the search tree, while the others contribute little or nothing to the process of finding a solution. Moreover, the communication is necessary for load balancing, since the local agenda of a processor may become empty if many non-expandable states have been reached (Kumar et al. 1994). Load balancing includes the transfer of states from one local agenda to another, in order to equalize the workload in all processors. This transfer can be performed directly or via a global memory structure, called blackboard.

There are two main problems related to both kinds (horizontal and vertical) of parallelization based on the distribution of the search space:

a) a great number of states is examined more than once, since the state-space is not always split in disjoined parts and
b) the states that are expanded are more than necessary.

Parallelization methods, which rely on horizontal distribution of the search space (e.g. IDA*), partially deal with problem (a), since different levels of the search space usually contain only a small number of states in common. However, these methods examine all the states at a given level before proceeding to the next one, causing problem (b). The alternative approach (vertical segmentation) suffers from both problems. The subtrees cannot be disjoined, since a state can usually be approached in various ways. Furthermore, a subtree might be promising (i.e. it contains a short solution), while the others are not, and yet the algorithm will examine all of them.

The latter problem becomes more severe as the heuristic function produces better estimates, since the set of promising states will become narrower and narrower. For example, if the heuristic function is perfect, a simple hill climbing technique will examine only l states, where l is the length of the optimal solution. Any one of the parallelization methods described previously will work N (number of processors) times more, since while one of the processors will be examining the solution's states, the others will be wasted at useless parts of the search space, or they will be re-examining the same promising states. Even if the accuracy of the heuristic estimate is less than 100%, but still acceptable, the overhead imposed by

the examination of redundant states would not allow the parallel algorithm to perform well.

3 The Operator Distribution Method

ODMP (**O**perator **D**istribution **M**ethod for Parallel **P**lanning) is a method for parallel planning, in which the task of finding and applying the applicable actions to a given state is performed in parallel. Suppose that we have M operators and N processors, where $M>N$. Initially, the operators are distributed to the available processors and then each processor is responsible for:

a) finding the applicable ground actions originating from the operators assigned to it,

b) applying the ground actions to the current state to produce the successor states and

c) evaluating the successor states through the heuristic function.

The distribution of the operators is done dynamically. Initially each processor is assigned one operator and the rest of them are stored in a global data structure, the *Operator Pool*, and are distributed on demand.

The distribution of the operators could be done statically at the beginning of the planning task. However, the amount of work imposed by each one of them cannot be accurately known a priori and the method would have resulted in an unfair distribution of work. The dynamic approach, which has been adopted by ODMP, succeeds in balancing the workload among processors, but imposes some overhead due to memory contention while accessing the *Operator Pool*. However, this overhead is negligible (as shown by the experimental results) compared to the speedup due to the balanced workload.

ODMP was motivated by the need for an effective parallelization method for heuristic, state-space planners, which usually exhibit the following characteristics:

a) They produce plans of relatively good quality, which means that parallelization methods that alter their solutions are usually undesirable.

b) The number of states they examine, in order to solve a problem, is relatively close to the length of the solution they produce. Since the states in the solution path have to be visited sequentially, the number of states that can be processed in parallel is quite limited. Therefore, parallelization methods that rely on the distribution of states are bound not to perform well.

Problem	Solution length (GRT)	Expanded states (GRT)	Solution length (CL)	Expanded states (CL)	Total states
Prob09	96	251	84	571	6.4 $*10^6$
Prob10	117	203	109	437	3.6 $*10^6$
Prob12	48	75	41	535	3.2 $*10^7$
Prob13	79	173	69	939	8.7 $*10^7$
Prob14	104	153	98	179	8.2 $*10^6$
Prob16	62	93	62	227	7.9 $*10^6$
Prob17	53	76	47	207	1.1 $*10^6$
Prob18	195	564	173	1035	5.4 $*10^7$
Prob19	174	515	157	898	3.0 $*10^7$

Table1: Number of states that are examined by the GRT and CL heuristic planners.

Table 1 illustrates the above statements for the two heuristic planners (GRT and CL) used for our experiments (a detailed description of these planners can be found in section 5). For the experiments we adopted a set of logistics problems (Veloso 1992) from the AIPS (International Conference on Artificial Intelligence Planning Systems) 1998 planning competition. Columns 2 and 4 present the lengths (number of actions) of the solutions that were produced by GRT and CL planners respectively. Columns 3 and 5 present the number of states that were expanded by GRT and CL planners respectively, in order to solve the problems and finally column 6 presents the total number of states in the search space of each problem. As we can see from table 1, the number of states that are examined by the planners is quite small with respect to the size of the search space (i.e. the number of examined states are approximately $1/10^5$ of the total states). Furthermore, the number of states that have to be examined is relatively close to the length of the solution (e.g. the number of examined states can be as low as 150% of the solution's length).

ODMP deals effectively with the characteristics of efficient heuristic planners. The quality of the solutions is not affected at all and no additional states have to be examined in order to solve the problem. Moreover, since the parallelization method does not depend on the number of states, the efficiency of ODMP is not affected by the quality of the heuristic function in use.

The parallelization is implemented using N of threads (usually equal to the number of available processors) that run simultaneously. The process that is created when the program is invoked is referred as the controlling thread. The controlling thread, whose algorithm is outlined in figure 1, is responsible for initializing, synchronizing and controlling the rest of the threads (denoted as planning threads). Figure 2 presents the algorithm that each one of the planning threads, which are used to solve the planning problem, executes. In figures 1 and 2 S_B denotes the current best state in the global agenda of the planner. N is the number of planning threads that will be created by the algorithm.

As mentioned before, the distribution of the operators is done dynamically. At the beginning of each cycle, all the domain's operators are copied in a shared data structure, the *Operator Pool*. When a planning thread has finished processing its current operator, it requests a new one from the *Operator Pool*, until the latter becomes empty. Apart from the *Operator Pool*, an additional one (the *Action Pool*) is used to parallelize the task of forming and evaluating the successor states. Parallelizing the tasks of finding the applicable ground actions and forming and evaluating the successor states in separate steps results in better load-balancing.

As presented in figures 1 and 2, the controlling thread updates the value of S_B only when all the planning threads have become inactive (step 5c in figure 1). This guarantees that S_B will always be assigned to the globally best state in the agenda and thus preserves the integrity of the original (sequential) planner, with respect to the produced plans.

1. Evaluate the *Initial State* using the heuristic function *h* and insert <*Initial State*, *h(Initial State)*> in the *Global Agenda.*
2. Set S_B=*Initial State.*
3. Create *N* planning threads *(N is user defined).*
4. Put all planning threads to sleep.
5. While *Goals* $\not\subseteq$ S_B do

    Begin

        5a. Put the domain's operators in the *Operator Pool.*

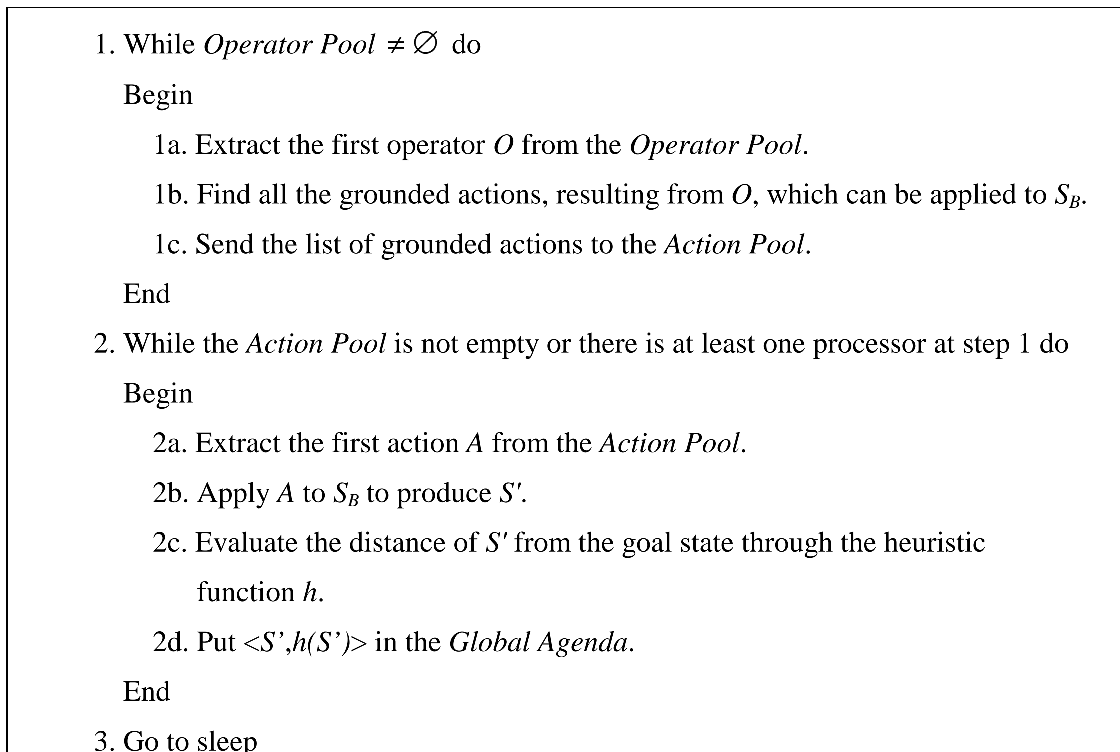
        5b. Awaken the planning threads.

        5c. Wait for all of the planning threads to become inactive.

        5d. Assign the best state in the *Global Agenda* to S_B.

    End
6. Return the Plan.

Figure 1. The algorithm of the Controlling Thread of ODMP



1. While *Operator Pool* $\neq \emptyset$ do

   Begin

   1a. Extract the first operator *O* from the *Operator Pool*.

   1b. Find all the grounded actions, resulting from *O*, which can be applied to S_B.

   1c. Send the list of grounded actions to the *Action Pool*.

   End

2. While the *Action Pool* is not empty or there is at least one processor at step 1 do

   Begin

   2a. Extract the first action *A* from the *Action Pool*.

   2b. Apply *A* to S_B to produce *S'*.

   2c. Evaluate the distance of *S'* from the goal state through the heuristic function *h*.

   2d. Put <*S'*,*h(S')*> in the *Global Agenda*.

   End

3. Go to sleep

Figure 2. The algorithm of the Planning Threads of ODMP

4 Extending the Operator Distribution Method

Although the previous algorithm results in a significant speedup of the planning process, it can be further improved by tackling three of its main inefficiencies, namely: a) the unbalanced distribution of the workload, b) the limitation in the scalability and c) the idle time due to memory contention.

The workload cannot be uniformly distributed to the available processors. The work imposed by each operator depends on the number of states that will be eventually produced by the application of this operator to the current state. Since this number can vary from zero to several hundreds, or even thousands for hard problems, depending on the current state, it is quite possible for some processors to stay idle for quite a long time waiting for the others to finish.

The scalability of the parallelization algorithm is limited by the number of the domains' operators. This number is usually small, since the common practice is to use general operators with many arguments for the encoding of the domains. For example, in the logistics domain, used in AIPS-98 and AIPS-00 planning competitions, there are only six operators: *Load-Truck*, *Load-Airplane*, *Unload-Truck*, *Unload-Airplane*, *Drive-Truck* and *Fly-Airplane*.

The processors stay idle for some period of time due to contention. In the algorithms presented in figures 1 and 2, there are three shared memory structures

(the *operator pool*, the *action pool* and the *global agenda*) that are accessed frequently by all processors.

In order to overcome these problems we have developed a number of techniques that can be considered as independent modules and can be added to the basic parallelization method presented in section 3. These techniques are the *Operator Reordering*, the *Semi-Grounded Operator Sets* and the *Distributed Agenda Architecture* that are presented in the following subsections.

4.1 Operator Reordering Technique

A number of Operator Reordering variations have been proposed as a means of changing the order in which the branches of the search space are traversed (Kumar et al. 1988, Cook et al. 1993). Inspired by these methods we studied the effect of operator reordering on the efficiency of ODMP. Our aim was not to reorder the search space, since ODMP uses a best first method that always expands the most promising state. However, a convenient order in the set of operators would result in a more balanced distribution of the workload.

Suppose we have N processors, M operators ($O_1,O_2,..,O_M$), each of which requires x *tu* (time units) on average to be processed, and another operator, denoted as O_{M+1}, which requires y *tu*. We also suppose that y is much greater than x and y is less than the time needed by the $N-1$ processors to process the operators $O_1,O_2,..,O_M$ in parallel ($y>>x$, $y<M*x/(N-1)$).

In the worst-case, O_{M+1} is placed last in the set of operators. After approximately $M*x/N$ *tu*, all the processors will be idle and O_{M+1} will be the only operator in the set. One processor will undertake O_{M+1} and there will be a period of y *tu*, where only one processor will be working and the remaining $N-1$ processors will stay idle.

In the best-case, O_{M+1} is placed first in the operators set. After y *tu* there will be $M-(N-1)*y/x$ operators in the set and the process will continue normally. Even if $y>M*x/(N-1)$, the rest $N-1$ processors will remain idle for $y-M*x/(N-1)$ *tu*.

We illustrate the previous example using concrete parameters: Suppose $N=10$, $M=100$, $x=5$ and $y=35$. In the worst-case scenario, the N processors would have worked in parallel for $x*M/N= 5*100/10= 50$ *tu* and one of them for another 35, resulting in a total execution time (for one iteration) of 85 *tu*. On the other hand, in the best-case scenario, after 35 *tu* the processors would have processed O_{M+1} and another 63 operators. After another 15 *tu*, there would be only 7 operators in the set. These 7 operators could be processed in parallel using 7 processors (3 would remain idle) in 5 *tu*. So the total execution time would be 55 *tu*.

According to our experience, the time spent for a given operator O_i is affected by the number of actions originating from it (denoted as $A(O_i)$), therefore a good method would be to place the operators in the set in a decreasing order of $A(O)$. However, for a given operator O_i the value of $A(O_i)$ depends on the state it has to be applied and it cannot be known a priori.

The *ODMP Operator Reordering Technique* is a simpler one, in which the operators are ordered once at the beginning and retain this order for the rest of the planning process. Since we cannot know a priori the actual value of *A(O)*, the ordering is done using the upper bound of *A(O)* instead.

For example, in a logistics problem with 5 cities, 7 airplanes, 5 trucks, 3 places per city and 2 cargoes, the maximum values of *A(O)* for the *Fly* and *Load_truck* operators are the following:

$A_{max}(Fly)$ = (7 airplanes * 4 possible target cities) = 28 and

$A_{max}(Load_truck)$ = (2 cargoes * 5 trucks) = 10.

4.2 The Semi-Grounded Operator Set Technique

Although the *ODMP Operator Reordering* technique described above succeeds in balancing the workload among the processors, the problem of the limitation in the scalability remains unsolved. In order to lift the bound in the scalability of our method, we developed an additional technique which works in a preliminary phase and increases the number of work packages that can be processed in parallel.

According to the *Semi-Grounded Operator Set* technique, the set of operators is expanded through the consideration of all the possible instantiations of the operators's first argument. We call the resulting operators *semi-grounded operators* and the expanded set *semi-grounded operator set*.

Consider, for example, a logistics problem with 3 cities (`city1, city2 and city3`), 1 plane (`A321`), 2 places per city (`center, airport`), 3 trucks (`truck1, truck2 and truck3`) and 4 cargoes (`cargo1, cargo2, cargo3 and cargo4`). The initial operator set includes the following six operators:

[*`Fly(A,S,D), Drive(T,S,D), Load_plane(C,A,L), Unload_plane(C,A,L), Load_truck(C,T,L), Unload_truck(C,T,L)`*]

where A, C, D, L, S and T are variables representing airplanes, cargoes, destination-locations, locations, source-locations and trucks respectively. The *semi-grounded operator set* contains one instantiation of the *fly* operator, three instantiations of the *drive* operator and four of each one of the other operators. For example, the three semi-grounded *Drive* operators are the following: [*Drive(truck1,S,D), Drive(truck2,S,D), Drive(truck3,S,D)*]. The total size of the expanded set is twenty.

The semi-grounded operator set contains a larger number of operators than the initial set, but the amount of work needed to process the two sets is the same, since the number of actions that will eventually be generated by these sets will be equal. So, the technique is capable of generating a relatively large number of disjoined segments of operators, which is equivalent to the initial operator set.

The efficiency of the parallel algorithm can benefit from this technique in several ways:

- The bound in the scalability is lifted, since the algorithm can utilize more processors.
- The workload is more balanced, since it is split in more portions.

The choice of the argument, which will be used to generate the semi-grounded operators, can affect the efficiency of ODMP. Due to the way in which the domains are encoded, the first argument is usually a rational choice and this approach is adopted by our algorithm.

4.3 Distributed Agenda Architecture Technique

In the case of applying ODMP to a heuristic planner relying on a best-first or a similar search strategy, an additional technique can be adopted that can drastically reduce the time wasted due to contention. According to this technique, the centralized agenda architecture proposed in figure 2 is replaced by a distributed architecture that minimizes the idle time due to contention, while preserving the integrity of the search method.

According to the *Distributed Agenda Architecture Technique*, apart from the global agenda, each processor maintains a local one where it stores all the states it has produced from the beginning of the planning process. At the end of each iteration, it extracts the best state from the local agenda and places it in the global one.

It is quite clear that this technique reduces the number of accesses to shared data, since the number of insertions in the global agenda at each iteration is N (the number of processors), where $N \ll b_F$ (the branching factor of the search space).

In order to prove that this technique maintains the integrity of the search method, it is enough to show that at each iteration the globally best state is considered for expansion. But since no states are pruned by the above technique, the globally best state at each iteration will be stored either in the global agenda or in a local one and it will be eventually promoted for expansion.

4.4 Assembling the Big Picture

In order to combine the above methods together and apply them to a planning system, certain issues have to be addressed:

The *Operator Reordering* technique and the creation of the *Semi-Grounded Operator Set* both work in preliminary phases and can cooperate in order to improve the efficiency of the parallelization. In order to optimize the algorithm, the operator set is initially ordered using the *Operator Reordering* technique, without taking into account the first argument of the operators. The output of this technique is then used to create the *Semi-Grounded Operator Set* that is used as input to the main planning algorithm.

The implementation of the parallel algorithm follows the common practice in parallel systems, i.e. there is a controlling thread that is responsible for the

preliminary steps and the management of the other threads, carrying out the main planning process. Figure 3 presents an overview of the controlling threads algorithm, while figure 4 outlines the algorithm run by the planning threads.

1. *Use the Operator Reordering technique to order the operators, without taking into account the operator's first argument.*
2. *Ground the operator's first argument to produce the Semi-Grounded Operator Set.*
3. *Insert <Initial State, h(Initial State)> in the Global Agenda.*
4. *Create N planning threads.*
5. *Initialize all the Local Agendas to* $\varnothing$.
6. *Set* S_B= *Initial State.*
7. *While Goals* $\not\subset S_B$ *do*

   Begin

   7a. Put the domain's operators in the Operator Pool.

   7b. Awaken the planning threads.

   7c. Wait for the planning threads to become inactive.

   7d. Assign the best state in the Global Agenda to S_B.

   End
8. *Return the Plan.*

Figure 3: The algorithm of the controlling thread (Extended ODMP)

1. *While the Operator Pool is not empty do*

   Begin

   1a. Request an Operator O.

   1b. Apply O to S_B *to create the successor states.*

   1c. Evaluate the successor states through the heuristic function.

   1d. Place the successor states among with their values in the Local Agenda.

   End
2. *Send the best state of the Local Agenda to the Global Agenda*
3. *Go to sleep*

Figure 4: The algorithm of the planning threads (Extended ODMP)

5 Planning Paradigms

In order to test the efficiency of ODMP in practice we embodied it in two heuristic planners, GRT Planner and CL planner, and run various experiments with the parallel programs. This section provides a brief description of the two planning paradigms.

5.1 The GRT Planner

The GRT planner is a domain-independent heuristic planner (Refanidis and Vlahavas 1999). It adopts the pure STRIPS representation (Fikes and Nilsson 1971) and searches forward in the space of the states. The planner has been inspired by the ASP planner (Bonet et al. 1997), but it has been differentiated in several ways.

GRT solves planning problems in two phases: the pre-processing phase and the search phase. The main idea of the planner is to compute off-line, in the pre-processing phase, estimates for the distances between the domains facts and the goals. The word 'distance' refers to the number of goal regression levels needed to achieve a specific fact. This information is stored in a table, which is indexed by the facts of the domain. This table is named GREEDY REGRESSION TABLE, which the acronym GRT comes from.

In order to produce better estimates, GRT introduces the notion of related facts in the goal regression process. These are facts that have been achieved either by the same or by subsequent actions, without the last action deleting the firstly achieved facts. The cost for achieving a set of un-related facts simultaneously is the sum of their individual costs, while the cost for achieving a set of related facts is the cost of the last achieved fact.

The search phase consists of a simple best-first search strategy. Based on the distances of the individual facts from the goals, and the information about their relations, GRT obtains estimates for the distances between the intermediate states and the goals, which are used to guide the search.

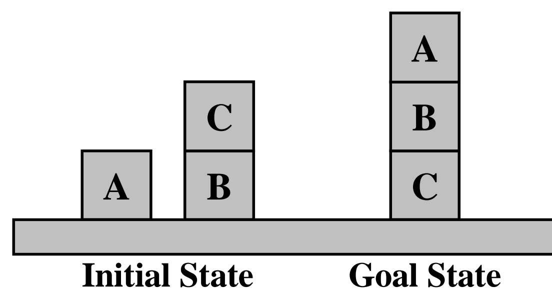


Figure 5: A 3-blocks problem

Fact	Distance from goals	Related facts
(on C table)	0	-
(on B C)	0	-
(on A B)	0	-
(clear A)	0	-
(on A table)	1	(clear B)
(clear B)	1	(on A table)
(on B table)	2	(on A table) (clear A) (clear B) (clear C)
(clear C)	2	(on A table) (clear A) (clear B) (on B table)
(on C B)	3	(on A table) (clear A) (on B table) (clear C)
...	...	...

Table 2: Part of the Greedy Regression Table
for the 3-blocks problem

5.1.1 An Example of GRT

We illustrate the GRT phases with the block world problem of figure 5. Part of the Greedy Regression Table for this problem is shown in table 2. Let us compute the distance between the initial state and the goals. The initial state consists of the following facts:

(*on A table*) (*clear A*) (*on B table*) (*on C B*) (*clear C*)

All the above facts are related, with the fact (*on C B*) being the last achieved. So the distance of all the facts is the distance of the last achieved, i.e. 3, which in this case is also the actual distance.

The above approach is followed to estimate the distances between all the intermediate states that arise during the forward search phase and the goals. GRT always selects to expand the state with the smallest estimated distance.

5.1.2 N-Best first search

GRT embodies a simple best-first algorithm and behaves very well in a variety of domains, including those used in AIPS-98. For the purposes of this research, we slightly modified the search algorithm and especially the agenda, in order to cope with more complex problems. The agenda in the current version has a limited size and the search algorithm is similar to the N-best-first used in one of ASP's versions (Bonet et al. 1997). Since the size of the agenda is kept under a threshold, the memory requirements of the modified GRT are quite low and thus GRT can handle even more difficult problems.

5.2 The Cargo Location Heuristic Planner

ODMP has also been tested on CL (Cargo Location) Planner (Vrakas et al. 2000), a simple best-first planner that uses a domain-dependent heuristic algorithm for logistics worlds. The planner estimates the distances between each intermediate state and the goals, taking into consideration the current locations and the destinations of the cargoes that have to be transferred.

To be more specific, for each cargo *c*, CL assigns an integer varying from 0 to 12, which represents the estimated number of steps for this cargo to reach its

destination (d_c). Then, the sum of these distances is the estimate for the distance between each intermediate state and the goals.

Initially each d_c is set to 0. Then the d_cs are computed by repeatedly applying for each cargo the following rules:

1. If c is not in its destination city, increase d_c by 4.
2. If c is not in its destination city and it is not in an airport, increase d_c by 4.
3. If c is not in its destination city and its destination place is not an airport, increase d_c by 4.
4. If c is in its destination city but it is not in the correct place, increase d_c by 4.

Consider, for example, the following case:

```
Goals≡     [at(c1,dc-ctr), at(c2,la-air),
          at(c3,la-ctr)]
StateA≡ [at(c1,dc-air), at(c2,dc-air),
          at(c3,dc-ctr)]
```

The distance between `StateA` and the `Goals` is estimated as follows:
Estimated distance between `StateA` and `Goals`

cargo1: 4 (4th rule)

cargo2: 4 (1st rule)

cargo3: 12 (1st, 2nd and 3rd rule)

Total = ***20***

6 Experimental Results

The two planners previously presented (GRT and CL) were implemented in C++ and were enhanced with ODMP and its extensions (Semi-Grounded Operator Set, Operator Reordering and Distributed Agenda Architecture) as described in section 4.4. For simplicity we will refer to GRT Planner with ODMP as *Parallel GRT* and to CL Planner with ODMP as *Parallel CL*. The two parallel planners were thoroughly tested on a variety of hard logistics problems, adopted by the AIPS-98 planning competition. The platform used for testing was a SGI Power Challenge XL parallel machine with 14 R8000 CPUs and 16 GB of shared memory. The underlying Operating System was IRIX 6.2.

Problem	Parallel GRT Planner						Parallel CL Planner					
	N=1	N=2	N=3	N=5	N=8	N=12	N=1	N=2	N=3	N=5	N=8	N=12
Prob09	140	84	61.1	40.2	30	29.7	254	144	110	70	52	55
Prob10	76	45	33	24.5	19.4	18.4	512	275	207	150	120.5	108
Prob12	123	66.5	48.6	32	23.6	20	2100	1200	900	600	450	370
Prob13	248	153	99	72	55	48.6	2660	1385	946	630	420	320
Prob14	212	133	96	67.5	51.7	42.4	265	152	120	80	57	49
Prob16	107	58	39.5	25.6	20.4	17.8	224	128	101	67	49	44.2
Prob17	51	32	22.7	15.3	12.1	11.3	118	72	50	33	28.1	33.7
Prob18	960	521.7	362.3	218.2	165.5	129.2	3900	2046	1415	855	570	450
Prob19	768	410.7	284.4	175	125.5	105	1350	800	523	366	270	220

Table 3: Time (in sec) needed by Parallel GRT and CL planners to solve the problems

Table 3 presents the time (in seconds) needed by Parallel GRT and Parallel CL to solve each one of the logistics problems. The columns correspond to different numbers (*N*) of utilized processors. In order to illustrate the efficiency of ODMP in more clarity, the speedup (T_{seq}/T_{par}) of the parallel planners was calculated and is presented in table 4. Figures 6 and 7 present graphs for illustrative examples of the speedup of Parallel GRT and Parallel CL planners respectively.

Problem	Parallel GRT Planner						Parallel CL Planner					
	N=1	N=2	N=3	N=5	N=8	N=12	N=1	N=2	N=3	N=5	N=8	N=12
Prob09	1	1.66	2.29	3.48	4.7	4.7	1	1.76	2.31	3.63	4.88	4.62
Prob10	1	1.69	2.3	3.1	3.92	4.13	1	1.86	2.47	3.41	4.25	4.74
Prob12	1	1.85	2.53	3.84	5.21	6.15	1	1.75	2.33	3.5	4.67	5.68
Prob13	1	1.62	2.5	3.44	4.5	5.1	1	1.92	2.81	4.22	6.33	8.31
Prob14	1	1.59	2.21	3.14	4.1	5	1	1.74	2.21	3.31	4.65	5.41
Prob16	1	1.84	2.7	4.18	5.25	6.01	1	1.75	2.22	3.34	4.57	5.07
Prob17	1	1.59	2.25	3.33	4.2	4.5	1	1.64	2.36	3.58	4.2	3.5
Prob18	1	1.84	2.65	4.4	5.8	7.43	1	1.91	2.76	4.56	6.84	8.67
Prob19	1	1.87	2.7	4.39	6.12	7.31	1	1.69	2.58	3.69	5	6.14

Table 4: Speedup of Parallel GRT and CL planners

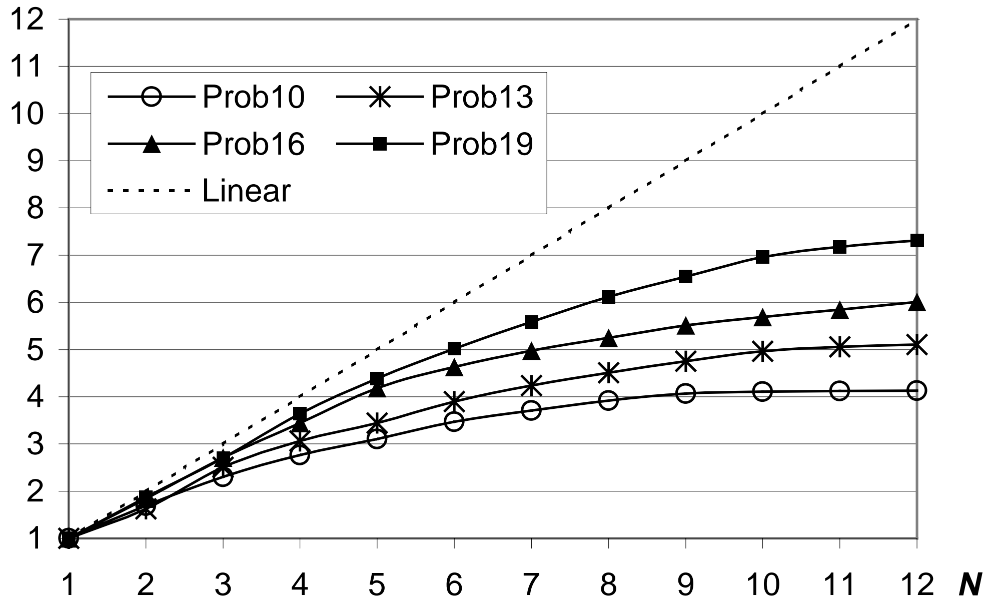


Figure 6: Speedup graph of Parallel GRT

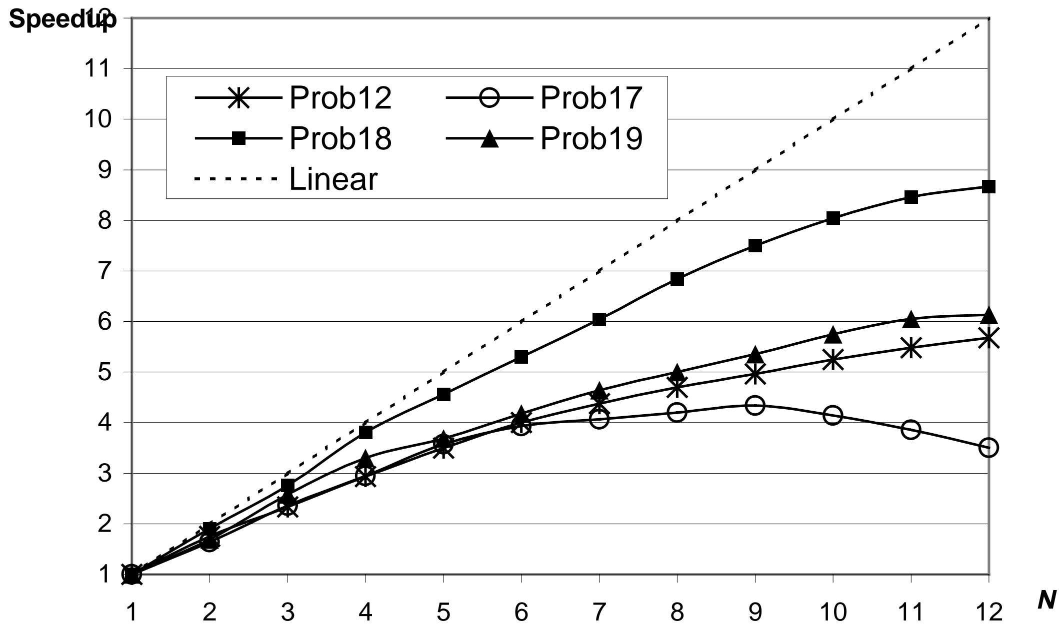


Figure 7 Speedup graph of Parallel CL

By analyzing the results presented in tables 3 and 4, we draw certain conclusions:

1. Under certain circumstances ODMP can speedup the underlying planner very efficiently. The speedup of Parallel CL is almost linear for Prob13 and Prob18 while Parallel GRT shows a quite stable speedup for Prob12, Prob16, Prob18 and Prob19.

2. There are cases where the speedup of the parallel planners drops off quite early (for example problems Prob10 and Prob17 for both parallel planners) although it performs quite well for low *N*s.
3. Although there are many differences in the heuristics of the two planning paradigms (Parallel GRT and Parallel CL) and they examine different set of states in order to solve the problems, yet in most cases the speedups of them are quite similar for specific problems. For example both planners speedup well in problems Prob12, Prob18 and Prob19 and they both fail to speedup well in problems like Prob10 or Prob17.

In order to justify the behavior of our method, we performed an analysis on the set of the logistics problems regarding their inner structure and its impact on our method. For each problem we computed the size of the *semi-grounded operator set* (denoted as SGOS), the upper limit of the number of grounded actions that can be applied to a given state (denoted as A_{max}) and the size of the search space, which indicates the complexity of the problem. Table 5 presents the results of our analysis.

Problem	**SGOS**	$\mathbf{A_{max}}$	**Size of the search space**
Prob09	80	204	$6.4 * 10^6$
Prob10	95	145	$3.6 * 10^6$
Prob12	84	409	$3.2 * 10^7$
Prob13	130	385	$8.7 * 10^7$
Prob14	171	185	$8.2 * 10^6$
Prob16	93	248	$7.9 * 10^6$
Prob17	110	135	$1.1 * 10^6$
Prob18	120	390	$5.4 * 10^7$
Prob19	117	318	$3.0 * 10^7$

Table 5. Number of semi-grounded operators and applicable actions

Comparing the values of A_{max} (table 5) and the *speedups* (table 4), we can conclude that the efficiency of ODMP depends strongly on A_{max}. This sounds quite reasonable, since the most resource-consuming part of the planning process, and the one that is performed in parallel, is the detection of the applicable actions and the formation of the successor states.

The scalability of ODMP and therefore its overall efficiency, is also affected by the number of semi-grounded operators (SGOS), since a low value of SGOS, means that there are not many work packages, the workload is not equally distributed and the overall performance decreases. For example, although A_{max} for Prob12 is the highest (409) among the adopted problems, the quite low value of

SGOS (84) prevents ODMP from performing well (e.g. speedup of CL for Prob12 and for N=12 is only 5.68).

Finally, the scalability of ODMP depends also on the complexity of the problem. Problems with large number of objects can be parallelized more efficiently. This makes sense, since the work imposed by each operator depends on the number of instantiations of its preconditions. Therefore, problems with operators that are easy to process do not allow the parallelization method to perform well, since a great portion of the time will be consumed in synchronization and communication among the threads. On the other hand, the parallelization method performs quite well on problems with operators that are hard to process.

7 Conclusion and Future Work

This paper reported on work performed to find an adaptive parallelization method for planning. We presented ODMP (Operator Distribution Method for parallel Planning), an innovative parallelization method that distributes the process of finding and applying the grounded applicable actions to a given state. ODMP can be adopted by any state-space planner, but is especially suited for heuristic planners, where the low number of examined states prevents classical parallelization methods from performing well.

Furthermore, we presented three techniques that extend ODMP and contribute to further improvement in the general performance. Namely these add-ons are: the *Operator Reordering Technique*, the *Semi-Grounded Operator Sets Technique* and the *Distributed Agenda Architecture Technique*. The extended ODMP has been thoroughly tested on GRT, a domain independent heuristic planner and CL, a domain dependent heuristic planner for Logistics problems. The experimental results were quite encouraging, since the method achieved a scale-up of over 8 on a parallel system with 12 processors for some problems.

This version of ODMP, although it performs well, is special crafted for parallel machines with shared memory. In the future, we plan to study the possibility of applying ODMP to parallel machines with distributed memory and, afterwards, in a network of computers. In order to do this, the parallelization method has to be altered, since the current version requires frequent intercommunication and the communication cost in networks of computers wouldn't allow this method to perform well.

8 References

[1] Blum, L., and Furst M., 1995, Fast planning through planning graph analysis, In *Proceedings, 14th International Joint Conference on Artificial Intelligence*, Montreal, Canada, pp. 636-642.

[2] Bonet, B., Loerincs, G., and Geffner, H., 1997, A robust and fast action selection mechanism for planning, In *Proceedings, 14th International*

Conference of the American Association of Artificial Intelligence (AAAI-97), Providence, Rhode Island, pp. 714-719.

[3] Cook, D., and Varnell, C., 1999, Adaptive Parallel Iterative Deepening Search. *Journal of Artificial Intelligence Research*, **9**: pp. 167-194.

[4] Cook, D., 1997, A Hybrid Approach to Improving the Performance of Parallel Search. In J. Geller (eds) *Parallel Processing for Artificial Intelligence* (Elsevier Science Publishers).

[5] Cook, D., Hall, L., and Thomas, W., 1993, Parallel search using transformation-ordering iterative-deepening A*. *The International Journal of Intelligent Systems*, **8** (8).

[6] Fikes, R., and Nilsson, N., 1971, STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, **2**: 189-208.

[7] Irani, K., and Shih, Y., 1986, Parallel a* and ao* algorithms: An optimality criterion and performance evaluation, In *Proceeding, 1986 International Conference on Parallel Processing*, St. Charles, Illinois, pp. 274-277.

[8] Kumar, V., Grama, A., and Rao, V., 1994, Scalable Load Balancing Techniques for Parallel Computers. *Journal of Parallel and Distributed Computing*, **22:** 60-79.

[9] Kumar, V., and Rao, V., 1990, Scalable parallel formulations of depth-first search. In Kumar, Gopalakrishnan and Kanal (eds) *Parallel Algorithms for Machine Intelligence and Vision* (Springer-Verlag), pp. 1-41.

[10] Kumar, V., Rao, V., and Ramesh, K., 1988, Parallel Best-First Search of State-Space Graphs: A Summary of Results, In *Proceedings, 7th National Conference on Artificial Intelligence*, St. Paul, Minnesota, pp. 122-127.

[11] Powley, C., Ferguson, C., and Korf, R., 1991, Parallel tree search on a simd machine, In *Proceedings, 3rd IEEE Symposium on Parallel and Distributed Processing*, pp. 249-256.

[12] Powley, C., and Korf, R., 1989, Single-agent parallel window search: A Summary of Results, In *Proceedings, 11th International Joint Conference on Artificial Intelligence*, Detroit, Miami, pp. 36-41.

[13] Rao, V., Kumar, V., and Ramesh, K., 1987, A parallel implementation of iterative deepening-A*. In *Proceedings, 6th National Conference on Artificial Intelligence*, Seattle, Washington, pp. 178-182.

[14] Refanidis, I., and Vlahavas, I., 1999, *GRT:* A Domain Independent Heuristic for STRIPS Worlds based on Greedy Regression Tables, In *Proceedings, 5th European Conference on Planning*, Durham, UK, pp. 346-358.

[15] Veloso, M., 1992, Learning by Analogical Reasoning in General Problem Solving, Ph.D thesis, CMU.

[16] Vrakas, D., Refanidis, I., Milcent, F., and Vlahavas, I., 1999, On the Parallelization of Greedy Regression Tables, In *Proceedings, 18th Workshop of the UK Planning and Scheduling Special Interest Group*, Manchester, UK, pp. 180-189.

[17] Vrakas, D., Refanidis, I., and Vlahavas, I., 2000, An Operator Distribution Method for Parallel Planning, In *Proceedings, AAAI Workshop on Parallel and Distributed Search for Reasoning*, Austin, Texas, pp. 17-21. `