In the Proceedings of the
18th Workshop of the UK
Planning and Scheduling SIG
December, 1999

On the Parallelization of Greedy Regression Tables

Dimitris Vrakas[1], Ioannis Refanidis[1], Fabien Milcent[2] and Ioannis Vlahavas[1]

(1)
Department of Informatics,
Aristotle University of Thessaloniki
54006, Thessaloniki, Greece
[dvrakas,yrefanid,vlahavas]@csd.auth.gr

(2)
Ireste School of Engineering,
University of Nantes
44306, Nantes, France
fmilcent@ireste.fr

Abstract

This paper presents PGRT, a parallel version of a best first planner based on the Greedy Regression Tables approach. The parallelization method of PGRT distributes the task of extracting applicable actions to a given state among the available processors. Although the number of operators limits the scalability of PGRT, it has proven to be quite efficient for low scale parallelization. A modified O*perator Reordering* method has been used in order to achieve further increase in the efficiency of the parallel algorithm. We illustrate the speedup of PGRT on a variety of hard logistics problems, adopted from the AIPS-98 planning competition.

1 Introduction

Heuristic functions are an important component of many artificial intelligence applications, especially when a “quite good” (not necessarily optimal) solution is required and there is a tight time limit. Planners are Artificial Intelligence applications, which given an initial state *I*, a set of possible actions and certain goals *G*, produce a plan of actions, which if applied to *I* achieves *G*. These programs are usually embedded in systems that must exhibit real-time behavior, so they are usually equipped with heuristic functions in order to respond promptly. Speed is the most desirable aspect of planning systems and although various methods, like hierarchical planning, case based planning, transformation to other problem types e.t.c., have been adopted, the absence of a good heuristic function makes a planning system inefficient for practical domains.

Recently, Refanidis & Vlahavas [16] introduced a new planner that is based on Greedy Regression Tables (GRT), a heuristic function for planning. GRT’s heuristic function is an enhancement of the simple and yet powerful idea used in ASP [2]. ASP calculates the summation of steps needed to achieve each goal independently, in order to estimate the distance between an intermediate state and the goals. GRT seems to outperform all the other known planners, such as GRAPHPLAN [1] and its ancestor ASP at least in some domains as the blocks world, the logistics and the gripper.

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 possibilities. Many researchers have tried to find parallelization techniques for AI applications and they have mainly focused on ways to distribute the search tree among the existing processors [3,4,10,12,13]. These techniques, which have been enriched with load balancing [9] and operator reordering [5,12], produce quite efficient parallel algorithms.

In this paper, we show that GRT examines only a small subpart of the search tree and thus methods relying on tree distribution cannot be applied efficiently to this planner. We present a different approach, which distributes the task of finding the grounded actions that can be applied to a given state. Each processor undertakes a number of operators and finds all the actions that are ground instances of these operators. The number of operators limits the scalability of PGRT, but it is very efficient for low scalability parallelization. Furthermore, we present a modified *Operator Reordering* method, which can achieve further increase in the efficiency of PGRT. This method changes the order in which the operators are processed, in order to balance the workload among the processors.

The rest of the paper is organized as follows: Section 2 presents previous work in the area of parallelization methods for AI problems. Section 3 briefly describes Greedy Regression Tables, while section 4 presents the modifications to the initial algorithm and outlines PGRT. Section 5 presents PGRT's performance results in comparison with the sequential version and section 6 introduces the modified Operator Reordering method and its effect on PGRT. Finally, section 7 concludes the paper and poses future directions.

2 Related Work

In [8], Kumar et al. 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 in [7], is that it does not result in much redundant search. However, the global agenda is accessed by all the processors very frequently and since it has to be protected by semaphores, the processors will stay idle for quite a long time.

On the other hand, in the distributed model each processor maintains its own local agenda and thus there is no need for semaphores. This model usually uses the IDA* search algorithm initially presented by Powley and Korf in [11]. 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. In [12] Powley and Korf introduced PWS, a tree distribution method in which each processor searches in a unique depth. Kumar et al., in [10] and [13] describe a different approach where the search tree is segmented vertically. To be more specific, 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. Moreover, Diane Cook in [3] and [4] proposed a hybrid approach, which combines IDA* and vertical segmentation techniques and seems to outperform all the other 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 examined [9]. 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. In [9], Kumar et al. review a number of receiver and sender initiated load-balancing techniques.

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 result in the expansion of more states than necessary. The first argument does not apply to IDA* since the search tree is split in almost disjoined parts, except for the states that can be approached by various ways of different length. However, IDA* examines all the states at a given level before proceeding to the next one (argument b). The alternative approach (vertical segmentation) suffers from both problems a and b. The subtrees can not be disjoined, since a state can usually be approached by different 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 was perfect, a simple hill climbing technique would have examined only *l* states, where *l* is the length of the optimal solution. Any one of the parallelization methods described previously would have worked *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. 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.

Since GRT's heuristic is quite accurate no one of the previous methods would have proven to be efficient. So in order to parallelize GRT, we need a different approach. In the next section we will present GRT in more detail, in order to show the parts of the algorithm that could be parallelized efficiently. Of course, the techniques described hereafter can be applied to any other planner equipped with a quite accurate heuristic algorithm.

3 Greedy Regression Tables

GRT (Greedy Regression Tables) is a new heuristic for planning proposed by Refanidis and Vlahavas [16], which improves the older ASP [2]. According to [16], ASP has two main inefficiencies: a) Each time a new distance has to be estimated a proposition graph similar to the one constructed by GRAPHPLAN [1] has to be reconstructed from scratch and b) it assumes that all facts can be achieved independently, not taking into account the interactions among them. GRT was initiated from the need to overcome these two drawbacks.

3.1 Estimating distances backwards

GRT works backward in order to estimate the distances between each fact in the domain and the goal state. The estimates are produced once at the preprocessing phase and they are used latter in the planning one. This feature overcomes ASP's first inefficiency, thus increasing the overall speed of the algorithm.

In order to compute the distances backwards the operators have to be inverted. Suppose we have a state S_1 and an action *a* that is applicable to S_1. Suppose also that S_2 is a successor state produced by applying *a* to S_1 (we note that as $S_2=res(S_1,a)$). The inverted action of *a, denoted as ~a*, is applicable to S_2 and $S_1=res(S_2,\sim a)$. Using STRIPS [6] terminology, *~a* can be constructed by *a* using the following formulas:

$$P(\sim a)=A(a)+P(a)-D(a)$$
$$D(\sim a)=A(a)$$
$$A(\sim a)=D(a)$$

where *P(X)*, *D(X)* and *A(X)* stand for precondition, delete and add list respectively. The heuristic uses the set of inverted actions and two algorithms presented in [16], in order to produce the Greedy Regression Tables in the pre-processing phase.

However, there is a considerable difficulty in the process of backward estimation because the set of goals in most cases does not form a complete state. For example, in the logistics problem only the final locations of the packages are specified and no information is available for the location of planes and trucks. This difficulty can be overcome if the set of goals is enriched with all the domain's facts that are not in contradiction with the goals. This process is done manually in the current version of GRT and therefore PGRT, but in [15] Refanidis et al. introduce certain methods, which can automatically enrich incomplete goal states.

3.2 Interactions among goals

As we stated earlier, ASP does not take into account the interactions among goals, but instead it assumes that the total number of steps needed to achieve a set of goals from a given state is the sum of the number of steps needed to achieve each goal separately. On the other hand, for each ground fact *p*, GRT keeps a list, denoted *rel(p)*, containing all the other facts that may also be achieved when achieving *p*.

3.3 PGRT's phases

GRT works in two phases: the pre-processing phase and the planning phase. At the preprocessing phase, the algorithm constructs the set of inverted actions and computes the enriched goal state. Then, each ground fact in the domain is assigned a distance equal to ∞ (dist=∞), except for those included in the enriched goal state, which are initialized to 0 and their *related lists* to ∅. The algorithm repeatedly applies the inverted actions to the enriched goal state, trying to achieve all the facts of the domain. At each iteration, the heuristic algorithms compute estimates for the distances of the newly achieved facts and construct their lists of related facts.

During the planning phase, GRT uses a simple Best-first algorithm that uses the distances and the related facts computed at the pre-processing phase to estimate the distances between any intermediate state and the goals.

3.3 N-Best first search

GRT has been embodied in a simple best-first algorithm and it has behaved very well in a variety of domains, including the ones used in AIPS-98. For the purpose 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 improved version has a limited size and the search algorithm is similar to the N-best-first used in one of ASP's versions. 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.

4 The Algorithm of PGRT

We performed various tests with GRT in different domains and we came to certain interesting conclusions:

i) The most resource consuming part of the algorithm is the detection of the actions that can be applied in a given state. Even with operator schemas, the work that has to be done is really hard, since there are thousands or even millions of grounded instantiations (actions) that have to be checked.

ii) The heuristic produces quite accurate estimates and as we present in Table 1, the number of examined states is relatively close to the length of the solution produced.

Logistics Problem (AIPS-98)	Solution Length	Expanded states
Prob09	98	252
Prob13	79	155
Prob14	104	149
Prob18	193	468
Prob19	174	413
Prob20	169	448
Prob21	120	318
Prob24	49	85

Table 1. Number of states expanded by GRT.

4.1 Overview

The parallel implementation of GRT was based on the previous conclusions. In PGRT the detection of ground actions that can be applied to a given state *S* is done in parallel. To be more specific, suppose that we have *M* operator schemas and *N* processors. We distribute the operator schemas to the available processors and each one will be responsible of finding the applicable ground actions originating from the schemas assigned to it.

The distribution can be done statically at the beginning; i.e. the first $\lceil M/N \rceil$ schemas will be assigned to the first processor, the next $\lceil M/N \rceil$ schemas to the second processor and so on. This approach is easy to implement and the overhead due to communication among processors is kept quite low. However, the number of ground actions originating from different schemas can vary from 0 to several hundreds (for a typical logistics problem) resulting in unbalanced workload among the different processors.

In the dynamic distribution method the unexamined operator schemas are kept all together in a global data structure, denoted as operator pool. Initially each processor is assigned one operator schema and the rest of the operators are sent on demand. This method can manage to balance the workload among processors, but imposes some overhead due to contention. However, this overhead is negligible compared to the speedup due to the balanced workload.

It is obvious from the previous description that the number of operator schemas limits the scalability of PGRT and therefore the current version is not suitable for massive parallelism.

4.2 Parallel algorithm

An outline of the algorithm running in each processor is presented in Figure 1. In this algorithm, S_B stands for the current best state in the global agenda.

```
1. While $S_B$ has not been defined, do nothing.
2. While operator pool is not empty:
        2a. Request an operator schema.
        2b. Find all the grounded actions
            that can be applied to $S_B$.
        2c. Send the list of grounded actions
            to the action pool.
3. While action pool is not empty or there
   is at least one processor at step 2:
        3a. Request new action.
        3b. Apply it to $S_B$ to produce S'.
        3c. Evaluate the distance of S' from
            the goal state using the
            heuristic function.
        3d. Send (S',dist(S')) to the global
            agenda.
4. Return to 1.
```

Figure 1. The main algorithm of PGRT

The first step of the parallel algorithm is used for synchronization between the various processors. The value of S_B will be updated only when all the processors have finished with the current iteration. This part is crucial, since if a processor was allowed to start a new iteration while the others are still working with the current one, S_B would be linked to a local best state that probably wouldn't be the global best one. The last one would have resulted in greater CPU usage, but also in larger number of examined states and consequently larger execution time.

In order to achieve further increase in the efficiency of the parallelization, the grounded applicable operators are temporarily stored in another pool (action pool) and the remaining tasks, i.e. creation of successor states and evaluation using the heuristic function, are done in an independent phase (step 3). This technique can offer further increase in CPU usage, since it contributes to better load balancing.

No synchronization is needed to control the transition from step 2 to step 3 and therefore a processor can proceed to step 3 while the others are still in 2.

5 Performance results

We have implemented PGRT in C++, using multithreading. Each thread contains the code corresponding to the algorithm presented in Figure 1 and communicates with the others through the shared resources (agenda, pools) and some global variables (S_B, flags). The various threads are controlled by a process, which acts as a normal thread (i.e. it contains the same code) but is also responsible of starting and stopping the threads and also of changing the value of S_B and making it available for use (step 1 of parallel algorithm).

According to [16], GRT solved the vast majority of the known planning problems in a few seconds time. The actual planning process needed considerable time only for some hard logistics problems, used in the AIPS-98, so in this paper we will focus on the performance of PGRT on these problems.

For the tests, we used two platforms, one for the estimated speedup as the number of processors increases and another one for the actual speedup with two parallel processors. The first one was a SUN ULTRA 1 workstation equipped with an 167 MHz processor and 64 MB of memory and the second one was a SUN ULTRA ENTERPRISE 3000 workstation with two processors at 167 MHz and 64 MB of memory. The operating system in both machines was SUN SOLARIS 2.51.

Table 2 illustrates the actual speedup of PGRT over GRT in a variety of hard logistics problems. The measurements were taken using the ENTERPRISE 3000 workstation. Columns 2 and 3 present the time (in seconds) spent by GRT and PGRT respectively to find a solution. The speedup presented in column 4 is calculated by the formula T_{GRT}/T_{PGRT} and its value lies between 1 and N (where N is the number of processors used).

Problem	T_{GRT}	T_{PGRT}	Speedup
Prob09	94	89	1.06
Prob13	185	149	1.24
Prob14	119	94	1.27
Prob18	662	393	1.68
Prob19	444	257	1.73
Prob20	521	288	1.81
Prob21	631	557	1.13
Prob24	245	175	1.4

Table 2. PGRT's speedup with two processors

Table 2 shows that for hard problems (Probs 18,19,20), PGRT can achieve a speedup of approximately 1.75 for two processors. One of the reasons that the speedup is less than two is that a portion of the execution time is spent for the pre-processing phase, which has not been parallelized.

Table 3 presents some estimates for the speedup of PGRT using more than two processors. These estimates were computed using statistical data acquired by a large number of tests performed on ULTRA 1. To be more specific, we ran many multithreaded versions of PGRT (using various numbers of threads) and measured the time spent for each thread, the time spent because of mutex locks, the time spent for unparallelized parts of the algorithm e.t.c. As we have stated earlier, the number of operators limits the scalability of our algorithm. Indeed in the logistics problem PGRT can be efficiently scaled up to 4 processors (the number of operators is 6).

Number of Processors	2	3	4	5
Speedup	1.75	2.25	2.5	2.45

Table 3. Estimated speedup

6 Operator reordering

Search methods, such as Iterative Deepening, Iterative Deepening A* or

Breadth First, search the nodes at a specific level from left to right. If the solution lies at the right end of the search tree, a significant increase at the speed of the search could have been achieved if the nodes at a specific level had a more convenient order. For example, a variation of the previous methods which search nodes from right to left could have found the solution sooner. However, since the exact location of a goal node is not known a priori, a more sophisticated method is necessary.

In [12], Powley and Korf propose two methods of ordering the search space. These two methods can be applied to IDA* search and reorder the nodes in the agenda, according to the *h* value of the heuristic function in use. An alternative method has been introduced by Cook et al. [5], according to which the set of operators is reordered in each iteration so as to guide the next IDA* search to the most promising node. The method proposed by Cook et al. can be easily implemented and according to performance results it can significantly increase the speed of search.

Inspired by the previous methods we studied the effect of operator reordering on the efficiency of PGRT. Our aim was not to reorder the search space, since PGRT uses a best first method which always expand the most promising state. However, a convenient order in the set of operator schemas could increase the efficiency of the parallelization. To be more specific, we claim that a specially selected order in the set of operator schemas would result in a more balanced distribution of workload among the existing processors.

Suppose we have N processors, M operators ($O_1,O_2,..,O_M$) each of which requires x seconds on average to be processed and another operator, denoted as O_{M+1}, which requires y seconds. We also suppose that y is 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 ($y>>x$, $y<Mx/(N-1)$).

In the worst case scenario, O_{M+1} is placed last in the set of operators. After approximately Mx/N seconds 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 seconds, where only one processor will be working and the rest $N-1$ will remain idle.

In the best case scenario O_{M+1} is placed first in the operator's set. After y seconds there will be $M-(N-1)y/x$ operators in the set and the process will continue normally. Even if we end up in a situation similar to the one in the first scenario, operator O_{M+1} will only require x seconds of processing.

We illustrate the previous example using concrete parameters: Suppose $N=10$, $M=100$, $x=5$ and $y=35$. In scenario 1, the N processors would have worked in parallel for xM/N= 5*100/10= 50 seconds and one of them for another 35, resulting in a total execution time (for one iteration) of 85 seconds. In scenario 2, after 35 seconds the processors would have processed O_{M+1} and another 63 operators. After another 15 seconds, there would be only 7 operators in the operator set. These 7 operators could be processed in parallel using 7 processors (3 would remain idle) in 5 seconds. So the total execution time would be 55 seconds.

In the previous analysis, we made some simplifications and therefore the actual difference between the two scenarios, in execution time, would be smaller. Furthermore, we didn't take into account step 3 of the parallel algorithm and thus the results we have made are overestimated. However, the total time spent for step 3 is very small compared to the time spent for step 2 and therefore the previous conclusions are quite accurate.

The time spent in step 2 for a given operator O_{i} is proportional to the number of actions originating from it (denoted as $A(O_i)$), therefore an ideal method would 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.

A simpler and therefore less efficient method of operator reordering have been applied to PGRT. According to this method, the operator schemas are ordered once at the beginning and retain this order for the rest of the planning process. Since we cannot know a priori the value of $A(O)$, the ordering is done using the maximum value 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 value of *A*(*O*) for each operator is the following:

- Fly: 7 airplanes * 4 possible target cities = 28.
- Load_plane: 7 airplanes * 2 cargoes = 14.
- Drive: 5 trucks * 2 possible target places = 10.
- Load_truck: 2 cargoes * 5 trucks = 10.
- Unload_plane, Unload_truck: 2 cargoes= 2.

We have tested the effect of the operator reordering technique on a variety of logistics problems and we illustrate the results in Table 4. In the third column we present the average speedup of PGRT over a large number of random orders. The last column presents PGRT's speedup using the best operators' order according to our convention. The default order in the second column is the one used in bibliography and AIPS-98 (it is the one also used in Table 2).

Problem	Default order	Random order	Best order
Prob09	1.06	1.04	1.10
Prob13	1.24	1.15	1.30
Prob14	1.27	1.15	1.37
Prob18	1.68	1.66	1.82
Prob19	1.73	1.70	1.81
Prob20	1.81	1.82	1.86
Prob21	1.13	1.20	1.27
Prob24	1.4	1.4	1.51

Table 4. Speedup of parallel algorithm

It is clear from Table 4 that the reordered operator set improves the efficiency of PGRT in all the tested problems. The difference from the default order is not significant in some problems, since the default order is quite close to the best one.

7 Conclusions and Future Work

This paper reports on work performed to find suitable parallelization methods for Greedy Regression Tables, a Best-First planner which was recently presented [16]. Many researchers, as Kumar, Cook and Powley, have proposed several parallelization methods that rely on the distribution of the search tree. However, we have shown that the heuristic function of GRT is quite accurate and only a small portion of the state space is expanded. Therefore the methods already proposed are unsuitable for GRT and any other accurate heuristic algorithm.

We proposed a different parallelization method, which distributes the process of finding the applicable actions among the available processors. Although the number of operator schemas limits the scalability of our approach, the results from various tests show that PGRT is quite efficient. We have also proposed an operator reordering schema, which can be easily incorporated in PGRT and offers a further increase in the performance of the parallel algorithm.

PGRT has been tested on a variety of logistics problems, taken from the AIPS-98 planning contest. In the future, we plan to test the efficiency and scalability of the algorithm on more complex domains with more operator schemas. Furthermore, we plan to adapt the concurrent heap, described in [14], for the global agenda in order to minimize the time spent due to contention.

As we stated earlier, the number of operators limits the scalability of PGRT and in the future we intend to lift this bound probably by using a cluster of processors for each operator. We could then achieve to increase the scalability of PGRT to several thousands (number of actions in a typical logistics problem).

Finally, we plan to develop methods for parallelizing the pre-processing phase of PGRT, which is currently executed sequentially. We expect to increase the speedup of PGRT by approximately 10% by parallelizing this phase, since it consumes a considerable amount of resources.

8 References

[1] L. Blum & M. L. Furst, "Fast planning through planning graph analysis", 14th International Joint Conference on Artificial Intelligence (IJCAI-95), Montreal, Canada 636-1642, 1995

[2] Bonet, G. Loerincs & H. Geffner, "A robust and fast action selection mechanism for planning", 14th International Conference of the American Association of Artificial Intelligence (AAAI-97), Providence, Rhode Island 714-719, 1997

[3] D.J. Cook and R.C. Varnell, "Adaptive Parallel Iterative Deepening Search", Journal of Artificial Intelligence Research, volume 9, pages 167--194, 1999.

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

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

[6] R. E. Fikes & N. J. Nilsson, "Strips: A new approach to the application of theorem proving to problem solving", Artificial Intelligence 2, 189-208, 1971

[7] K. B. Irani and Y. F. Shih, "Parallel a* and ao* algorithms: An optimality criterion and performance evaluation", Proceedings of international Conference on Parallel Processing, pages 274-277, 1986

[8] V. Kumar, V. N. Rao and K. Ramesh, "Parallel Best-First Search of State-Space Graphs: A Summary of Results (1988)", Proceedings of the 1988 National Conf. on Artificial Intelligence (AAAI-88), 1988.

[9] V. Kumar, A. Y. Grama and V. N. Rao, "Scalable Load Balancing Techniques for Parallel Computers", Journal of Parallel and Distributed Computing, Volume 22, Number 1, pp. 60-79, 1994.

[10] V. Kumar & V. N. Rao, "Scalable parallel formulations of depth-first search", In Kumar, Kanal & Gopalakrisham (Eds.), Parallel Algorithms for Machine Intelligence and Vision, pp. 1-41 Springer-Verlag, 1990.

[11] C. Powley, C. ferguson and R. E. Korf. "Parallel tree search on a simd machine", Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing, pages 249-256, 1991

[12] C. Powley and R. E. Korf, "Single-agent parallel window search", IEEE Transactions on Pattern Analysis and Machine Intelligence, 13(5), 1991.

[13] V. N. Rao, V. Kumar & K. Ramesh, "A parallel implementation of iterative deepening-A*", Proceedings of the National Conference on Artificial Intelligence, pp. 178-182. Morgan Kaufmann, 1987.

[14] V. N. Rao & V. Kumar, "Concurrent Insertions and deletions in a priority queue", Proceedings of the 1988 Parallel Processing Conference, 1988.

[15] I. Refanidis, I. Vlahavas and L. Tsoukalas, "*On Determining and Completing Incomplete States in STRIPS Domains*", IEEE International Conference on Information, Intelligence and Systems, Washington D.C., 1999 (to be presented).

[16] I. Refanidis and I. Vlahavas, "*GRT: A Domain Independent Heuristic for STRIPS Worlds based on Greedy Regression Tables*", 5th European Conference on Planning (ECP-99), Durham, UK, Springer-Verlag, 1999 (to be published).