A Suitable data structure for parallel A

Van-Dat Cung, B. Le Cun

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

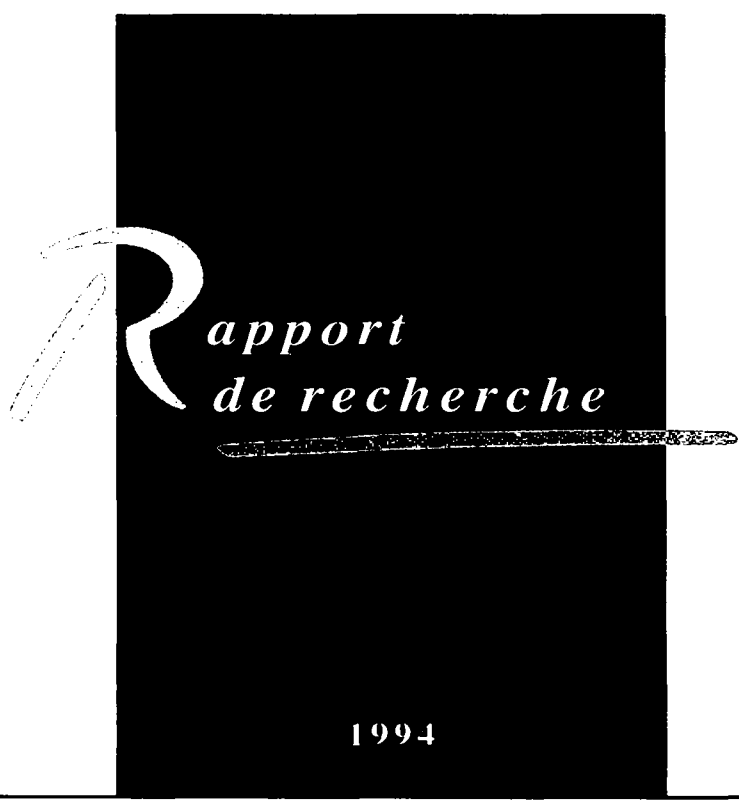
*A Suitable Data Structure for Parallel A**

Van-Dat CUNG, Bertrand LE CUN

N° 2165
Janvier 1994

PROGRAMME 1

Architectures parallèles,
bases de données,
réseaux et systèmes distribués



A Suitable Data Structure for Parallel A*

Van-Dat CUNG* and Bertrand LE CUN*

Programme 1 — Architectures parallèles, bases de données, réseaux et systèmes distribués

Abstract: This report presents a new parallel implementation of the heuristic state space search A* algorithm. We show the efficiency of a new double criteria data structure called *treap*, instead of usual priority queues (heaps). This data structure allows operations such as *Insert*, *DeleteMin* according to one criterion and *Search* on the other criterion. These operations are essential in the A* algorithm.

Furthermore, we give concurrent algorithm of the treap within a shared memory environment. Results on the 15 puzzle are presented; they have been obtained on two machines, with virtual or not shared memory, the KSR1 and the Sequent Balance 8000.

Key-words: Heuristic search, state space traversal, A*, data structure, binary search tree, priority queue, parallelism, concurrence.

*May also be contacted at the Laboratory PRiSM, Université de Versailles - St. Quentin en Yvelines 45, Avenue des États-Unis, F-78000 Versailles, France. E-mail: van-dat.cung@prism.uvsq.fr / bertrand.lecun@prism.uvsq.fr or Van-Dat.Cung@inria.fr / Bertrand.Le_Cun@inria.fr

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : (33 1) 39 63 55 11 – Télécopie : (33 1) 39 63 53 30

Une Structure de Donnée Adaptée au A* Parallèle

Résumé : Ce rapport présente une nouvelle implantation parallèle de l'algorithme de parcours heuristique d'espace de recherche, A*. Nous montrons l'efficacité d'une nouvelle structure de données à double critères, appelée *treap*, quant aux habituelles files de priorité (heaps). Les opérations d'insertion (*Insert*), de suppression du minimum (*DeleteMin*) suivant un critère et de recherche (*Search*) suivant l'autre critère peuvent être effectuées sur cette structure. Elles sont essentielles dans l'algorithme A*.

Nous avons également rendu ces opérations de la treap concurrentes pour un environnement à mémoire partagée. Les résultats sur le taquin 4x4 sont présentés. Ils sont obtenus sur deux machines, l'une avec mémoire distribuée mais virtuellement partagée (KSR1), l'autre avec mémoire partagée (Sequent Balance 8000).

Mots-clé : Recherche heuristique, parcours d'espace d'états, A*, structure de données, arbre de recherche binaire, file de priorité, parallélisme, concurrence.

1 Motivations

Search is a technique widely used in Artificial Intelligence (AI) and Operational Research (OR) for solving Discrete Optimization problems [18, 17, 19, 26]. The space of potential solutions of these problems can be specified, but the difficulty is that its cardinality is too large to be enumerated (time exponential in the size of the problem instance). Such problems are Combinatorial Optimization problems (as Traveling Salesman Problem, Quadratric Assignment Problem, ...) or Artificial Intelligence games (as 15 puzzle, ...). Indeed, many of these problems have been classified as NP-complete and search is one of the few available means for solving them.

The space of potential solutions of these problems is generally defined in AI in terms of state space. A state space is defined by:

1. an initial description of the problem called *initial state*,
2. a set of operators that transform one state into another,
3. a termination criterion which is defined by the properties that the solutions or the *set of goal states* must satisfied.

This state space is often huge but not untractable. Thus, parallelism is logically an idea for speeding up the traversal of this space. It could reduce the searching time and therefore increases the size of problems solved. Clearly, if many processors are available, then they can search different parts of the space concurrently. Research in this field is very active [11, 9, 14, 20, 7].

However, a straightforward parallelization of state space search may not be efficient. For many problems, heuristic domain knowledge is available and is gathered during the traversal of a state space. This knowledge can then be used to avoid searching some useless parts of the state space. If the state space is divided statically into disjoint parts which are distributed between different processors, this may eventually do more work than a single processor. The extra work may definitively reduce the speedup that can be obtained by parallel processing. The fairness of data structures used in a parallel implementation is also a key factor for the concurrency of access.

In this paper, we propose a new asynchronous parallel implementation of the A* state space algorithm. It differs mainly from the few previous work [11] in the data structures used to implement the OPEN and CLOSED lists of the A* algorithm, respectively a priority queue plus a hash table. We present a more suitable data structure called *treap* for the A* algorithm. Operations have access to the data structure according to two different criteria. The first one allows the access to the treap as a priority queue, the second one as a binary search tree.

This new data structure solves problems such as deadlocks and concurrency of access. The usual combined data structures (priority queue plus hash table) are absolutely not suitable because of cross-references. By combining two criteria into one data structure, the treap suppresses cross-references and makes the concurrent access management much more easier. Thus better performances should be obtained.

This implementation is done on shared memory architectures. New massively parallel machines like the KSR1 propose a *virtual* shared memory mechanism, it is interesting to test and compare its effectiveness in the case of A* with respect to other parallel machines with a *classical* shared memory such as the Sequent Balance 8000.

2 Heuristic search A*

The implicit representation of the state space is commonly defined as a weighted, directed graph. Each state is represented by a vertex and each transformation from one state to another is represented by a directed edge. Here after, we call without distinction a vertex for a state and vice-versa. The weight of an directed edge is the cost $c(v_i, v_j)$ of generating a new state v_j by applying the corresponding operator on state v_i. Let us denote v_0 the vertex corresponding to the initial state and v_n a vertex corresponding to a goal state. Thus, to find a optimal solution in the state graph is equivalent to find a least cost directed path from the initial state v_0 to the set of goal states.

The A* algorithm [18, 19] is a well-known heuristic search for finding a least cost path between an initial state and goal states of a given state graph. The A* algorithm maintains two lists called OPEN and CLOSED. The OPEN list contains those vertices whose successors have not yet been generated, and the CLOSED list those vertices whose successors have been generated.

Each state in the graph is assigned a cost with an evaluation function f, the cost is denoted by f-value. The traversal strategy of the state graph is a *best-first* strategy according to the f-values. At each iteration of the algorithm, a state with the best f-value in the OPEN list is selected for expansion.

In the A* algorithm, the evaluation function f (cf. Figure 1) is the sum of two other functions g and h: $f = g + h$.

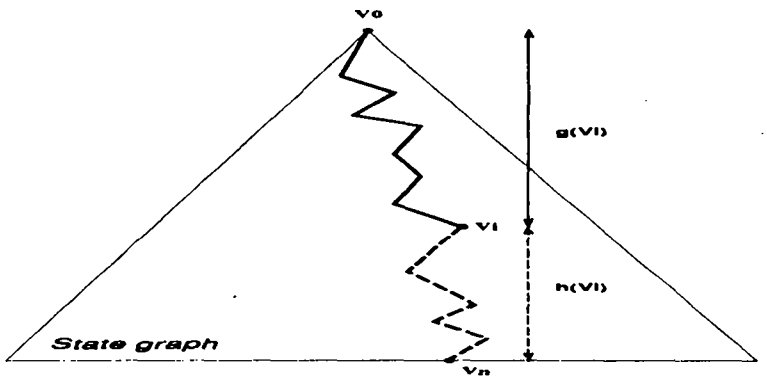


Figure 1: Evaluation function f.

Definition 1 *Let v_0 be the initial state, v_n a goal state and v_i a current state of the search. The functions g and h are defined as follows:*

- $g(v_i)$ *is the cost of the current path from the initial state* v_0 *to the current state* v_i,
- $h(v_i)$ *is a* heuristic *function which estimates the cost of the optimal path from* v_i *to a goal state* v_n*; if* v_i *is a goal state then* $h(v_i)$ *is equal to 0.*

At the beginning of the search, OPEN contains only the initial state v_0 and its corresponding f-value $f(v_0)$. At each iteration, the A* algorithm selects the most promising state according to its f-value. This state becomes the current state v_i. If this state is a goal state or if no more state is available in OPEN, the algorithm would respectively terminate with v_i as solution or proclaim a failure. Otherwise, the current state is expanded by applying the operators which are defined on v_i. Once the state v_i has been expanded, it is removed from OPEN and added to CLOSED. For each successor v_j of v_i, if state v_j has not been generated before, it is added to the OPEN list. If state v_j has been generated before and its new f-value is greater than its current f-value in OPEN or CLOSE then v_j is discarded; otherwise, the previous f-value of v_j is substituted with the new one. If the f-value of state v_j has been updated and v_j is in CLOSED, then transfer the state v_j from CLOSED to OPEN.

This search process continues until a goal state v_n is found or the OPEN list becomes empty.

Several properties of the A* algorithm can be pointed out. They are all founded on two assumptions:

1. the cost $c(v_i, v_j)$ of each edge from state v_i to state v_j is greater than ϵ strictly positive,
2. the heuristic function h is always positive.

Under these assumptions, three main properties [18, 19] hold; the first two concern the termination of the algorithm and the third one the update of states in the CLOSED list. We recall them briefly.

Property 1 : *The A* algorithm would terminate on finding a goal state if there is one (A* is complete).*

Property 2 : *If the heuristic function* h *is* admissible, *that is,*
if $h(v_i) \leq h^*(v_i)$ *for all state* v_i *where* $h^*(v_i)$ *is the cost of the optimal path from state* v_i *to a goal state* v_n*;*
then the A algorithm is guaranteed to terminate with a least cost path from the initial state to a goal state, if there is one (A* is admissible).*

Property 3 : *If the heuristic function* h *is admissible and monotone, that is,*

$$0 < h(v_i) - h(v_j) \leq c(v_i, v_j), \quad \textit{for all successors } v_j \textit{ of } v_i,$$

then the values given by the evaluation function f *are increasing,*

$$f(v_i) \leq f(v_j), \quad \textit{for all successors } v_j \textit{ of } v_i;$$

The property 3 implies that no state in the CLOSED list will be updated and transferred to the OPEN list for future re-expansion. Under this condition, the delete operation in the CLOSED list becomes useless. However, we should not discard the CLOSED list, because otherwise we would re-expand some state already expanded and do redundant work.

On the performance evaluation aspect, the A* algorithm always maintains a search tree during the traversal of a state graph.

Definition 2 *For a given search tree of a state space traversal, we call:*

- heuristic branching factor b, *the average number of successors, over all the search tree, that are generated by the application of an operator to a given state;*
- depth d, *the length or the number of the applied operators used to transform the initial state into a goal state of the least cost path.*

It has been shown [10] that the average time complexity of A* is $O(b^d)$ and the average space complexity is also $O(b^d)$. These results lead us to think that parallelism may help us to speedup the search and by this way to solve problems of larger sizes.

3 Parallelization of A*

Several parallelizations have been proposed for the Branch and Bound procedure with *best-first* traversal strategy (see [7] for a large survey). But at out best knowledge, there are only a few specific parallel implementations of the A* algorithm [11, 5] which is in fact a Branch and Bound like procedure with dominance relation [19, 17]. Moreover, recent works [24, 21, 9, 22] in this area seem to prefer parallelizing the IDA* algorithm to the A* one.

The Iterative-Deepening A* algorithm [10] is essentially, on the opposite of the A* algorithm, a *depth-first* search procedure combined with the technique of iterative deepening. At each iteration, IDA* performs a bounded (thresholds) depth-first search of the state graph by using an admissible monotone evaluation function f.

The reason of this preference is that the A* algorithm seems to be more difficult to parallelize than IDA*, from a point of view of simplicity and overheads [24]. Thresholds in the IDA* algorithm could be searched in parallel [9]. Whereas the only work to do in parallel in A* is the management of OPEN and CLOSED global lists. Furthermore, no suitable data structures for the OPEN and CLOSED lists have been designed in parallel formulations given in the literature.

A simplest parallelization of the A* algorithm is to let all available processors work on one of the current best state in the OPEN list, following an asynchronous concurrent scheme. Each processor gets work from the global OPEN list. This scheme has the advantage that it provides small search overheads, because global information are available for all processors [11] via the OPEN list. This scheme is particularly well suit for multiprocessors with shared memory.

However, parallel processing introduces two difficulties. First, the termination criterion of the A* algorithm does not work any more. We can not insure that the first goal state found is the optimal one. The optimal solution could be concurrently computed by another processor. Thus, we propose that termination occurs only after a goal state has been found by one processor, and when no any other processor has a better f-value state to expand.

Secondly, since the global OPEN and CLOSED lists are access asynchronously by every available processor, contention for the lists will greatly limit the performances. But this drawback could be reduced if the data structures could be accessed concurrently. Thus we propose to use a *concurrent treap* (see paragraphs 4 and 5) for OPEN and a *hash table* for CLOSED.

We also introduce two improvements in the A* algorithm.

The first one is the *local best state* notion. After expanding a state, instead of adding all the successors in the OPEN list, each processor compares locally the f-value of each newly generated state with the one of the best state in OPEN (i.e. the root of the treap).

If the f-value of the best state in OPEN is better than the one of a newly generated state, this state is added to OPEN. Otherwise, this state is kept by the processor and becomes the local best state for this processor. This notion would save a lot of useless operations (*Insert* and *DeleteMin*) on OPEN, because successors of a state have often better f-values than the one of the best state in OPEN.

The second improvement concerns the tests in the CLOSED list. In the A* algorithm, before adding a new state in OPEN we have to verify if this state exists already in CLOSED. Otherwise, because of the asynchronous concurrent scheme, it may induce contention in the CLOSED list. Since a selected state from OPEN is added to CLOSED after its expansion, we propose to check CLOSED only when a state is selected for expansion. This will reduce the number of access to the CLOSED list in parallel.

We implemented this asynchronous concurrent scheme for solving the 15 puzzle [10] which could almost be considered as a benchmark for A*-like algorithms. The game consists of a 4x4 square frame containing 15 square tiles and an empty position. The operator slides any tile adjacent to the empty position into this position. The goal of the game is to rearrange the tiles from an initial configuration (state) into a desired goal configuration. The Figure 2 shows an example of a 3x3 puzzle.

As the previous works, we use the Manhattan distance heuristic for the function h. This distance is defined as follows:

Definition 3 *Let A and B two positions of a square frame with coordinates (x_A, y_A) and (x_B, y_B), the Manhattan distance between A and B is*

$$d(A,B) = |x_A - x_B| + |y_A - y_B|.$$

However, in case of equal f-values in OPEN, the state with the smallest value of h is selected for expansion [22]. Thus, we have modified a little bit the evaluation function f in order to take in account this tie-breaking rule. Our evaluation function is now $f = C(g+h)+h$, where C is a constant equal to $10^{\lceil \log_{10}(2(n-1)(n^2-1)) \rceil}$, with n the size of a

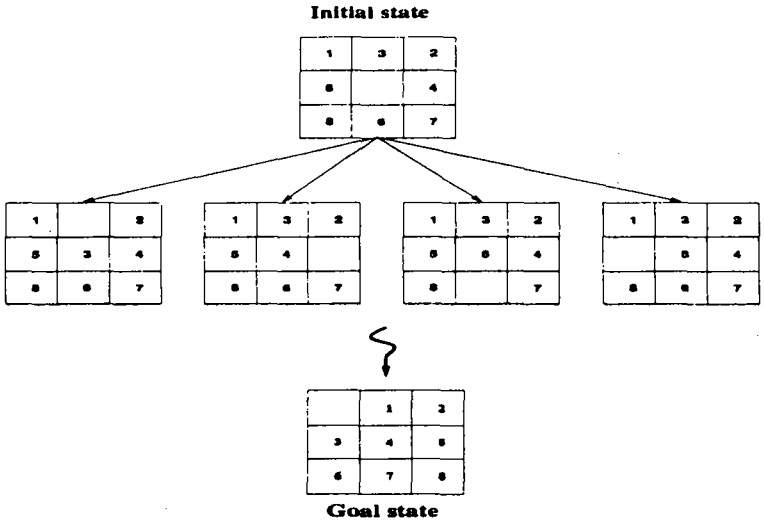


Figure 2: 3x3 puzzle.

$n \times n$ square frame. For the 15 puzzle, the number n is equal to 4 and the number C is equal to 100.

Let us point out that the evaluation function $g + h$ is monotone, but it is no longer the case for $C(g + h) + h$. Thus, updated states in the CLOSED list could be transfer to the OPEN list for re-expansion.

4 Efficient data structures

In the A* algorithm, the operations we need for the OPEN list are essentially *Insert*, *DeleteMin* and *Search*. Those of the CLOSED list are *Insert*, *Delete* and *Search*. The *Insert* operations consist of adding an element in a list (OPEN or CLOSED), while the *Search* operations identify if a state already exists or not in one of the two lists.

As the A* algorithm uses the *best-first* search strategy, it explores at each iteration the best f-value state in the OPEN list. This is accomplished by the *DeleteMin* operation. The *Delete* operation removes a state from the CLOSED list. It is used when a updated state is transferred from CLOSED to OPEN.

Priority queues [1, 28] may be suitable for the *DeleteMin* operation according to the f-value criterion, they are completely inefficient for the *Search* operation with the state criterion. Other data structures such as hash table, AVL-trees [1], splay-trees [6, 29], etc, are efficient for the *Search* operation but not for *DeleteMin*. There were no data structures with both operations with respective criterion. This is the reason why the data structure used for OPEN is usually a combination of a priority queue (heap) and a hash table, and the one for CLOSED is a hash table. The hash table is relatively easy to implement and

the access can be concurrent without any problem. In contrast, concurrent access to a priority queue are not simple to achieve.

Furthermore, the parallelism increases some specific problems as synchronization overheads, because operations have to be done in an exclusive manner. The combination of two data structures for the OPEN list also implies cross-references and thus the deadlock problem.

Thus, we propose a new data structure called *treap* which combines the two criterions (the f-value for *DeleteMin* and the state for *Search*) into one structure. This suppressed cross-references and limits synchronization overheads.

In the next sections we discuss more precisely this data structure and how implement concurrent access to it.

5 Treap Data Struture

Treap was introduced by Aragon and Seidel [2]. The authors use them to implement a new form of binary search tree : Randomized Search Trees. McCreight [16] uses them also to implement a multi-dimensional searching. He called it *Priority Search Tree*[1].

Let X be a set of n items, a *key* and a *priority* are associated to each item. The keys are drawn from some totally ordered universe, and so are the priorities. The two ordered universes need not to be the same.

A *treap* for X is a rooted binary tree with node set X that is arranged in In-order with respect to the keys and in Heap-order with respect to the priorities.

In-order means that for any node x in the tree $y.key \leq x.key$ for all y in the left subtree of x and $x.key \leq y.key$ for all y in the right sutree of x. *Heap-order* means that for any node x with parent z the relation $x.priority \leq z.priority$ holds.

It is easy to see that for any set X such a treap exists. The item with the largest priority is in the root node.

The A* algorithm uses an OPEN list where a *key* (a state of the problem) and a priority (an evaluation of this state) are associated to each item. Thus, we can use a treap to implement the OPEN list of the A* algorithm.

Let T be the treap storing set X. The operations presented in the literature that could be apply on T are *Search*, *Insert* and *Delete*. We add one more operation *DeleteMin* and modify the *Insert* operation to implement the OPEN list of A*. Here are the definitions and the properties of our set of operations :

- $Search(T, key)$ finds the item x in T such that $x.key = key$;
- $Insert(T, x)$ adds the item x in T but $x.key$ must be unique in T;
  let y be an item already in T such that $y.key = x.key$,

[1] Vuillemin introduced the same data structure in 1980 and called it *Cartesian Tree*. The term *treap* was first used for a different data structure by McCreight, who later abandonned it in favour of the more commonly used *priority search tree* [16].

- if $y.priority < x.priority$, x is inserted, y is removed,
- if $x.priority \leq y.priority$, x is not inserted;

- $DeleteMin(T)$ selects and removes the item x in T with the highest priority;
- $Delete(T, key)$ removes the item x from T such that $x.key = key$.

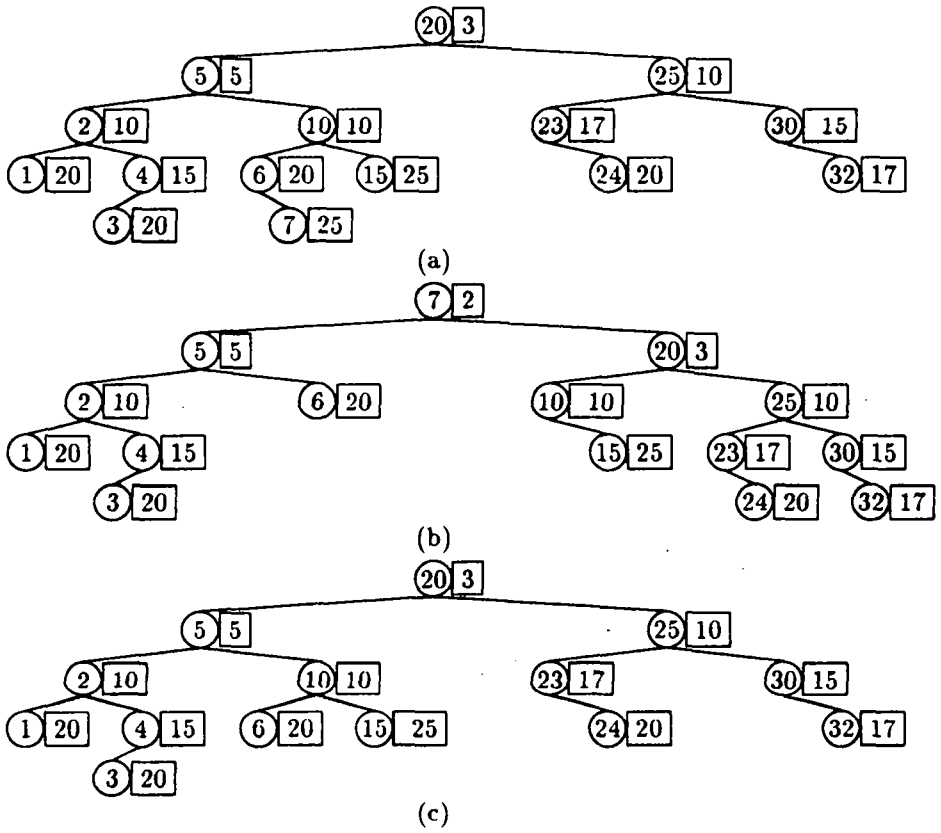


Figure 3: (a)→(b) Insert(7,2) − (b)→(c) DeleteMin

5.1 Sequential operations

We explain the different sequential operations on the treap.

Given the key of x, an item $x \in X$ can be easily accessed in T by using the usual search tree algorithm.

As several binary search trees [27, 29, 6, 12, 4], the update operations use a basic operation called Rotation (Figure 4).

In the literature, the *Insert* operation is as follows. At first, using the key of x, attach x to T in the appropriate leaf position. At this point, the key of all the node of the modified

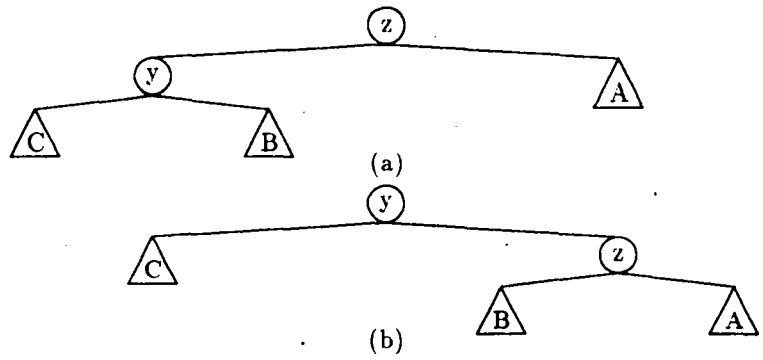


Figure 4: Rotation used to reorganized the treap.

tree are in In-order. To re-establish Heap-order, simply rotate x as long as its parent have a smaller priority.

To keep the properties of the *Insert* operation as defined above, the *Insert* algorithm can not be used in this form. We design a new algorithm which inserts an item x in T.

Using the key of x, we search the position, with respect to the In-order and the Heap-order. That is, we use the *Search* algorithm but it stops when:

- if a item y is found such that $y.priority < x.priority$,
- if an item z is found such that $z.key = x.key$.

If such an item z is found, the algorithm ends, because it is garanteed that $x.prority < z.priority$, thus x must not be inserted in T. However, if such an item y is found, the item x is inserted at this position (between y and the father of y).

Let SBT_x be the subtree rooted in x of T. At this point, the priorities of all the node of the modified tree (SBT_x) are in Heap-order. To re-establish the In-order we use the splay operation [27, 29]. SBT_x is splitted in a Left subtree and a Right subtree. The left subtree (resp: right) contents all the items of SBT_x with the key smaller (resp: bigger) than the key asociated with x. Finally left subtree and the right subtree are attached to x. Then, SBT_x and T are in In-order and in Heap-order.

If we find an item y such that $y.key = x.key$, during the splay operation, the item y, is deleted (the y priority will be smaller then x priority).

The *DeleteMin* and *Delete* operations are not very different. The *DeleteMin* operation removes the root of T, and the *Delete* operation removes the root x of a subtree of T such that $x.key = key$. Thus, first we search such a node x and we apply a *DeleteMin* on the subtree rooted in x.

The *DeleteMin* operation is achieved as follows. Let x be the root of the treap T. We rotate x down until it becomes a leaf (where the decision to rotate left or right is dictated by the relative order of the priorities of the children of x), and finally clip away the leaf.

Each node contains one item (a key and a priority), thus, the set occupies $O(n)$ words of storage.

The time complexity of each operation is proportionnal to the depth of the treap T. If the key and the priority associated with an item are in the same order, the structure is a linear list. However, if the priorities are independant, identically distributed continuous random variables, the depth of the treap is $O(logn)$ (the treap is a balanced binary tree). Thus, the expect time to perform one of these operations, is still $O(logn)$ (n number of node in T) [16].

To get an balanced binary treap, in a implementation for the A* algorithm, the problem is *reversed*. The priority order can not be modified. However, we can find an arbitrary bijective function to encode the ordered set of keys into a new set of randomized ordered keys. The priority order and the key order are then different.

5.2 Concurrent operations

In asynchronous parallel A* algorithm implemented on multiprocessors with shared memory, the exclusive use of the entire treap to perform a basic operation serializes the access to the treap. But the speed-up obtained with the parallel algorithm is limited.

Each operation on the treap manipulates the data structure in the same *Top-Down* direction and is made of successive elementary operations. We can use the technique denoted by **Partial Locking** [15, 4, 8, 23] to reduce the contentions. Each processor holds exclusive use of the smallest subset of needed items. Hence, the time delay, during which the access to the structure is blocked, is decreased.

The *tree partial locking protocol* uses the paradigm of *user view serialization* introduced by Lehman and Yao, 1981 [13], Calhoun and Ford, 1984 [3]. Every processor in concurrent implementation sees and modifies the data structure as if it could hold the entire tree excluding access. To hold an exclusive access on each node of the treap, a locking protocol or marking protocol (boolean like) is used.

If each processor respect a well ordering scheme to lock the node, this technique allows us to maintain consistency of the data structure and to avoid deadlock. All the proof and several details can be found in [15, 4].

The Treap implementation on a Sequent Balance use the partial locking with the marked protocol. On the KSR1, the subpage primitives are used to implement the Locking protocol on each node.

6 Empirical results

Here we discuss our experimental results on the KSR1 and the Sequent Balance 8000 in the context of the 15 puzzle problem.

The KSR1 is a massively parallel architecture machine which can scaled up to 1088 processors. Our machine have 32 processors, each of them has 32 megabytes of local

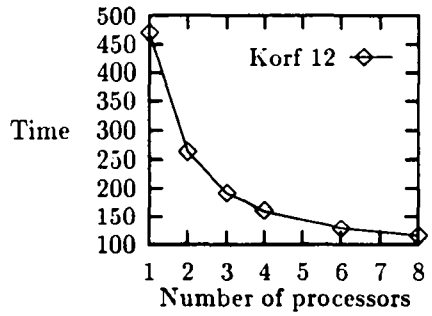


Figure 5: Results on the Sequent Balance 8000.

memory. The local memories are shared between the processors through the ALLCACHE system which emulates a global shared memory. When a processor needs data which are not in its local memory, it asks the ALLCACHE system to search in and copy the corresponding data from the other processors.

The Sequent Balance 8000 is a classical parallel machine with respect to the KSR1. We have 10 processors and 16 megabytes of global memory. The memory is shared between processors via fast hardware locks. If a processor needs to write data in the memory, it first locks the corresponding cells for exclusive access. Thus, there is no copy of data in the global memory.

We selected four instances of the 15 puzzle presented in [10] according to their difficulties. The total number of states generated varied between approximatively 60 thousand and 1,5 million. The running times are presented in Figures 5 and 6. The figure 7 shows the corresponding speedups.

On the running time curves, we remark that times reduce quickly according to the number of processors used on every instance of the problem. This is verified for both machines. However, we have not been able to compute big instances on the Sequent Balance because of memory lack. Only the smallest instance has been computed.

On the same instance of the problem, we note that the KSR1 is faster than the Sequent Balance, but the speedup obtained by the second one is better (see Figure 7). The fact that the Sequent Balance has a real shared memory explains the latter results. The ratio communication speed over processor speed is greater on the Sequent Balance than on the KSR1.

We also observe a speedup anomalie for the instance Korf78 when 2 and 3 processors are used. This happens because in parallel A*, a goal state could be found earlier with several processors than in sequential [24, 25].

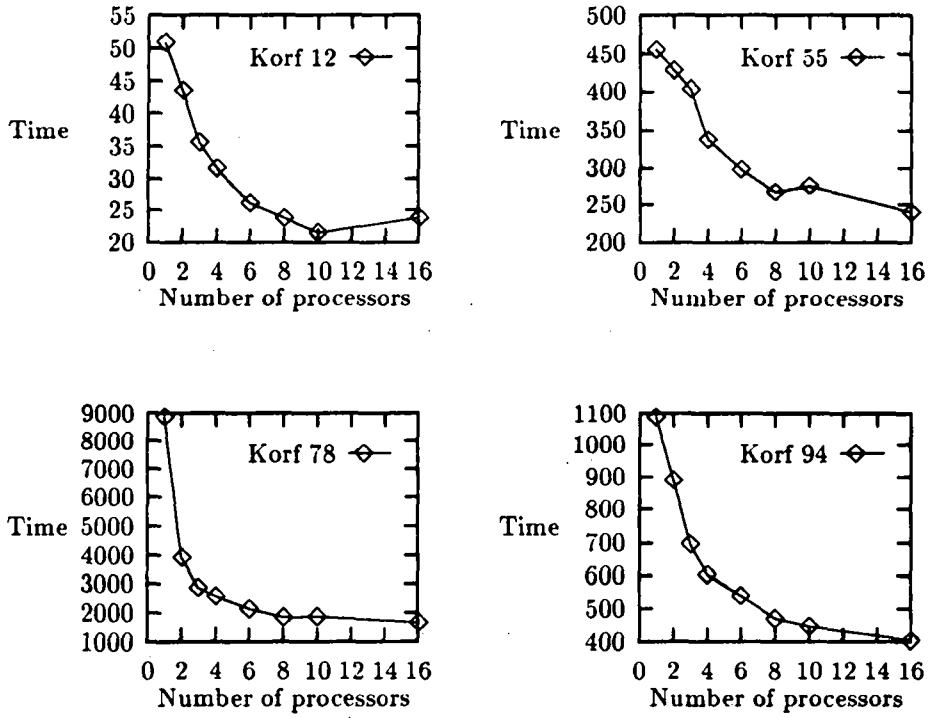


Figure 6: Results on the KSR1.

The speedup differences between the instances are essentially due to the sizes of the problem. The instance Korf78 is significantly large with respect to the others.

For the 15 puzzle problem, the expansion of a state and the evaluations of its successors need a few amount of time with respect to the access time on the data structures. That is the reason why speedup curves are not linear to the number of processors used. Applying this parallel A* algorithm on another problem where the local computation time is greater, better speedups could be obtained.

7 Concluding remarks

We have presented in this paper a specific parallel implementation of the A* algorithm for the 15 puzzle problem on two shared memory machines.

A new data structure, the treap is proposed with the operations *DeleteMin* and *Search*. This data structure allows us to store a set of items containing a pair of key and priority. We can apply on the same data structure the basic operations of binary search trees and those of priority queues.

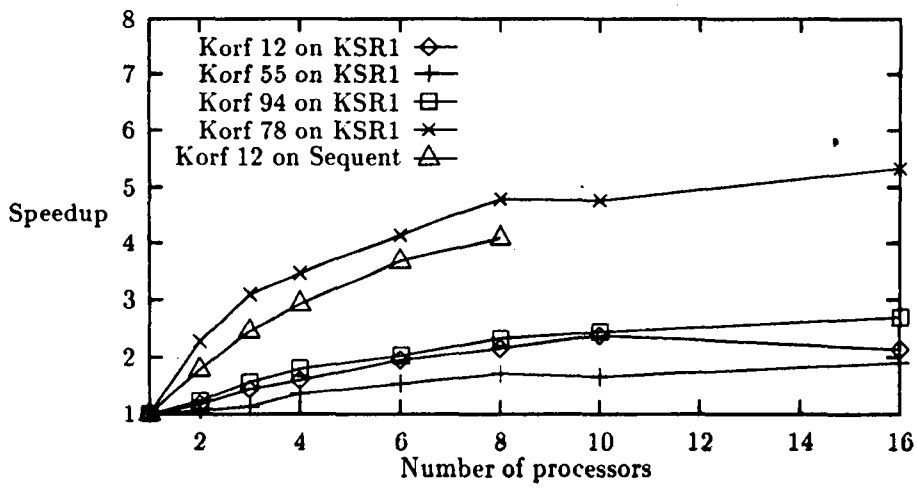


Figure 7: Speedups.

To have basic operations of binary search trees and priority queues applied to the same data structure is essential to implement efficiently the A* algorithm.

We have implemented concurrent access for these operations with the technique called Partial Locking. This technique could be easily apply to tree data structures.

Results obtained on both machines (KSR1 and Sequent Balance 8000) show that the treap is efficient for a group of ten processors. The speedups presented are better than those in [24] with the same parallel scheme.

The difference between the speedups curves on both machines lets us conclude that the ratio communication speed over processor speed is greater on the Sequent Balance than on the KSR1. This leads us to implement currently a version of parallel A* algorithm with more data locality for the OPEN and CLOSED lists on the KSR1.

References

[1] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[2] C. Aragon and G. S. R. Randomized search trees. *FOCS 30,* pages 540–545, 1989.

[3] J. Calhoun and R. Ford. Concurrency control mechanisms and the serializability of concurrent tree algorithms. In *of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, Waterloo Ontario, Apr. 1984. Debut de la theorie sur la serializability.

[4] B. L. Cun, B. Mans, and C. Roucairol. Opérations concurrentes et files de priorité. RR 1548, INRIA-Rocquencourt, 1991.

[5] V.-D. Cung and C. Roucairol. Parcours parallèle de graphes d'états par des algorithmes de la famille a* en intelligence artificielle. RR 1900, INRIA, Apr. 1993. In French.

[6] C. Ellis. Concurrent search and insertion in avl trees. *IEEE Trans. on Cumputers*, C-29(9):811–817, Sept. 1981.

[7] A. Y. Grama and V. Kumar. A survey of parallel search algorithms for discrete optimization problems. Personnal communication, 1993.

[8] D. Jones. Concurrent operations on priority queues. *ACM*, 32(1):132–137, Jan. 1989.

[9] L. Kalé and V. A. Saletore. Parallel state-space search for a first solution with consistent linear speedups. *International Journal of Parallel Programming*, 19(4):251–293, 1990.

[10] R. E. Korf. Depth-first iterative-deepening : An optimal admissible tree search. *Artificial Intelligence*, (27):97–109, 1985.

[11] V. Kumar, K. Ramesh, and V. N. Rao. Parallel best-first search of state-space graphs : A summary of results. *The AAAI Conference*, pages 122–127, 1987.

[12] H. Kung and P. Lehman. Concurrent manipulation of binary search trees. *ACM trans. on Database Systems*, 5(3):354–382, 1980.

[13] P. Lehman and S. Yao. Efficient locking for concurrent operation on b-tree. *ACM trans. on Database Systems*, 6(4):650–670, Dec. 1981.

[14] A. Mahanti and C. J. Daniels. Simd parallel heuristic search. Technical Report UMIACS-TR-91-41, CS-TR-2633, Computer Science Department, University of Maryland, College Park, Maryland, May 1991.

[15] B. Mans and C. Roucairol. Concurrency in priority queues for branch and bound algorithms. RR 1311, INRIA-Rocquencourt, Oct. 1990.

[16] E. M. McCreight. Priority search trees. *SIAM J Computing*, 14(2):257–276, May 1985.

[17] D. S. Nau, V. Kumar, and L. Kanal. General branch and bound, and its relation to a* and ao*. *Artificial Intelligence*, 23:29–58, 1984.

[18] N. J. Nilsson. *Principles of Artificial Intelligence.* Tioga Publishing Co., 1980.

[19] J. Pearl. *Heuristics.* Addison-Wesley, 1984.

[20] C. Powley, C. Ferguson, and R. E. Korf. Parallel tree search on a simd machine. In *The Third IEEE Symposium on Parallel and Distributed Processing*, Dec. 1991.

[21] C. Powley and R. E. Korf. Simd and mimd parallel search. In *The AAAI symposium on Planning and Search*, Mar. 1989.

[22] C. Powley and R. E. Korf. Single-agent parallel window search. *IEEE Transactions on pattern analysis and machine intelligence*, 13(5):466–477, May 1991.

[23] V. Rao and V. Kumar. Concurrent insertions and deletions in a priority queue. *IEEE proceedings of International Conference on Parallele Processing*, pages 207–211, 1988.

[24] V. N. Rao, V. Kumar, and K. Ramesh. Parallel heuristic search on shared memory multiprocessors : Preliminary results. Technical Report AI85-45, Artificial Intelligence Laboratory, The University of Texas at Austin, June 1987.

[25] C. Roucairol. Recherche arborescente en parallèle. RR M.A.S.I. 90.4, Institut Blaise Pascal - Paris VI, 1990. In French.

[26] C. Roucairol. Exploration parallèle d'espace de recherche en recherche opérationnelle et intelligence artificielle. In M. Cosnard, M. Nivat, and Y. Robert, editors, *Algorithmique parallèle*, Études et recherches en informatique, pages 201–211. Masson, 1992. In French.

[27] D. Sleator and R. Tarjan. Self-adjusting trees. In *15th ACM Symposium on theory of computing*, pages 235–246, Apr. 1983.

[28] D. Sleator and R. Tarjan. Self-adjusting heaps. *SIAM J. Comput.*, 15(1):52–69, Feb. 1986.

[29] R. Tarjan and D. Sleator. Self-adjusting binary search trees. *Journal of ACM*, 32(3):652–686, 1985.

Unité de Recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

Unité de Recherche INRIA Lorraine Technopôle de Nancy-Brabois - Campus Scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 VILLERS LES NANCY Cedex (France)
Unité de Recherche INRIA Rennes IRISA, Campus Universitaire de Beaulieu 35042 RENNES Cedex (France)
Unité de Recherche INRIA Rhône-Alpes 46, avenue Félix Viallet - 38031 GRENOBLE Cedex (France)
Unité de Recherche INRIA Sophia Antipolis 2004, route des Lucioles - B.P. 93 - 06902 SOPHIA ANTIPOLIS Cedex (France)

EDITEUR
INRIA - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

ISSN 0249 - 6399