Parallel and Distributed Branch-and-Bound/A Algorithms*

V.-D. Cung, S. Dowaji, B. Le Cun, T. Mautor and C. Roucairol

N° 94/31

10 octobre 1994

Parallel and Distributed Branch-and-Bound/A* Algorithms*

V.-D. Cung[†], S. Dowaji, B. Le Cun[†], T. Mautor and C. Roucairol[†]

Thème/Theme 1 — Parallèlisme/Parallelism
Équipe/Team PNN[‡]

Rapport de recherche n ° 94/31 — 10 octobre 1994 — 25 pages

Abstract: In this report, we propose new concurrent data structures and load balancing strategies for Branch-and-Bound (B&B)/A* algorithms in two models of parallel programming : shared and distributed memory.

For the shared memory model (SMM), we present a general methodology which allows concurrent manipulations for most tree data structures, and show its usefulness for implementation on multiprocessors with global shared memory.

Some priority queues which are suited for basic operations performed by B&B algorithms are described : the *Skew-heaps*, the *funnels* and the *Splay-trees*. We also detail a specific data structure, called *treap* and designed for A* algorithm. These data structures are implemented on a parallel machine with shared memory : KSR1.

For the distributed memory model (DMM), we show that the use of partial cost in the B&B algorithms is not enough to balance nodes between the local queues. Thus, we introduce another notion of priority, called *potentiality*, between nodes that takes into account not only their partial cost but also their capacity to generate other nodes. We have developed three load balancing strategies using this potentiality.

The B&B implementation was carried out in a network of heterogeneous Unix workstations used as a single parallel computer through the Parallel Virtual Machine (PVM) software system.

In order to test the efficiencies of concurrent data structures and load balancing strategies for B&B and A* algorithms, we implemented them for solving Combinatorial Optimization Problems (COPs) as Quadratic Assignment Problem (QAP), Vertex Cover Problem (VCP) and puzzle game.

Key-words: Data structures, priority queues, concurrency, load balancing, parallel and distributed algorithms, Branch-and-Bound, A*, PVM.

(Résumé : tsvp)

[‡]Email : pnn@prism.uvsq.fr

*This work was partially supported by the projet Stratagème of the french CNRS.

[†]May also be contacted at *INRIA, Domaine de Voluceau, Rocquencourt, BP105, 78153 Le Chesnay Cedex, FRANCE.*

Laboratoire PR*i*SM
Université de Versailles - Saint Quentin en Yvelines
78035 Versailles Cedex, FRANCE.
Téléphone/Phone : (33 1) 39 25 40 56 – Télécopie/Fax : (33 1) 39 25 40 57

Algorithmes Branch-and-Bound/A* Parallèles et Distribués[‡]

Résumé : Nous proposons, dans ce rapport, des nouvelles structures de données concurrentes et stratégies d'équilibrage de charges dédiées aux algorithmes Branch-and-Bound[§] (B&B)/A* dans deux modèles de programmation parallèle : mémoire partagée et distribuée.

Pour le modèle avec mémoire partagée (SMM), nous presentons une méthodologie générale permettant les manipulations concurrent sur la plupart des structures de données arborescentes, et nous montrons son efficacité dans les implantations sur les multiprocesseurs avec mémoire partagée globale.

Certaines files de priorité répondant aux exigences des opérations de base des algorithmes B&B sont décrites : les "Skew-heaps", les "funnels" et les "Splay-trees". Nous détaillerons également une structure de données spécifique, appelée "treap" et conçue pour l'algorithme A*. Toutes ces structures de données sont implantées sur une machine parallèle avec mémoire partagée : la KSR1.

Pour le modèle avec mémoire distribuée (DMM), nous montrons que l'usage du coût partiel (borne inférieure) dans les algorithmes B&B n'est pas suffisant pour répartir équitablement les sommets entre les files locales. Ainsi, nous introduisons une artre notion de priorité entre les sommets, nommée "potentialité", qui prendre en compte non seulement leurs coûts partiels mais aussi leurs capacités à engendrer d'autres sommets. Nous avons développé trois stratégies d'équilibrage de charges utilisant cette potentialité.

L'implantation de l'algorithme B&B est faite sur un réseau hétérogènes de stations d'UNIX. Ce réseau est utilisé comme une seule machine parallèle à l'aide de l'environnement "Parallel Virtual Machine" (PVM).

Afin de tester les efficacités des structures de données concurrentes et les stratégies d'équilibrage de charges pour les algorithmes B&B et A*, nous les avons appliqués à la résolution des problèmes d'optimisation combinatoire (COPs) tels que l'Affectation Quadratique (QAP), la Couverture Minimale (VCP), et le jeu du Taquin.

Mots-clé : Structures de données, files de priorité, concurrence, équilibrage de charges, algorithmique parallèle et distribuée, Branch-and-Bound, A*, PVM.

[‡] Ce travail est partiellement supporté par le projet Stratagème du CNRS.

[§] "Évaluation et Séparation" en français.

Contents

1 Introduction **1**
1.1 COPs and Branch-and-Bound/A* formulations 1
1.2 Data structures and programming models 2

2 Concurrent access to data structures **3**
2.1 Partial locking protocol 3
2.2 Partial locking boolean protocol 4
2.3 Related works 4
2.4 Priority queues for best-first B&B 6
2.4.1 Experimental results 8
2.5 Concurrent Treap for parallel A* 9
2.5.1 The A* algorithm 9
2.5.2 Towards a double criteria data structure 10
2.5.3 Treap data structure 11
2.5.4 Experimental results 14

3 Load balancing strategies **15**
3.1 The PVM environment 15
3.2 Distributed best-first B&B 16
3.2.1 B&B for VCP 16
3.2.2 Implementation 16
3.3 Motivations and justifications of our approach 16
3.4 Results with classical priority notion 18
3.5 A new notion of priority: potentiality 18
3.6 Load balancing strategies with potentiality 19
3.6.1 Influence on load distribution 20
3.6.2 Influence on communication costs 20

4 Concluding remarks & perspectives **21**

1 Introduction

Processing larger problems in a shorter time has always been one of the main challenges in computer science.

This point is particularly important for the solution of Combinatorial Optimization Problems (COPs) which require high processing speed and a lot of memory space. Numerous scientific, military and industrial applications are expressed as COPs (e.g. VLSI circuit design, production control, resource allocation, command system, robots, etc).

Massively parallel processing, which has been developed to satisfy these needs, is therefore a natural choice. The challenge is how to take advantage of the huge processing capacity provided by a large number of processors in terms of performance (speed-up and/or scale-up). Parallel algorithm development (design, analysis and implementation) is thus the price to pay to reach the expected results.

1.1 COPs and Branch-and-Bound/A* formulations

Formally, a COP in its minimization release can be defined as follows: Given a finite discrete set X, a function $F : X \rightarrow \mathbb{R}$, and a set S where $S \subseteq X$, find an *optimal* solution $x^* \in S$, such that $F(x^*) = \min\{F(x) | \forall\, x \in S\}$. The set S is called domain of constraints and all the elements $x \in S$ are *feasible* solutions. We assume that S is finite or empty.

For example, in Operational Research (OR), the Quadratic Assignment Problem (QAP) and the Vertex Cover Problem (VCP) fit the above definition. In Artificial Intelligence (AI), solving a puzzle game is generally stated as finding a shortest path from an initial configuration to the final configuration. All these COPs are *NP-complete* or *NP-hard*.

To find an exact solution to a COP, the best way consists of using a Branch-and-Bound (B&B) paradigm including B&B algorithms in OR and A* algorithm in AI [34]. The B&B paradigm explores - cleverly - the set X. The knowledge acquired along the search path is used to discard the exploration of the useless parts of X. Synthetically, B&B paradigm could be summed up as follows:

Building the search tree

- a *branching scheme* splits X into smaller and smaller subsets,
- a *search strategy* selects one node among all pending nodes to be developed.

Pruning branches

- a bounding function $f : 2^S \rightarrow \mathbb{R}$ gives a *lower bound* for the value of the best solution in each set S_i of S created by branching,
- the *exploration interval* is revised periodically, as the upper bound is constantly updated every time a feasible solution is found;
- in certain applications, *dominance relationships* may be established between subsets S_i, and will thus also lead to discard non-dominant nodes[1].

[1] The A* algorithm uses this kind of relationships for graph search.

From an algorithmic point of view, a B&B algorithm carries out a sequence of basic operations on a set of elements with their priority value :

- *DeleteMin* selects and deletes the highest priority node,
- *Insert* adds a new node with precomputed priority,
- *DeleteGreater* deletes nodes with lower priorities than a given value which is usually the upper bound.

In practice, *priority queues* are usually chosen as suitable data structures for implementing these operations.

1.2 Data structures and programming models

Unlike the Numerical Computation which generally manipulates regular and static data structures (e.g. arrays), B&B/A* algorithms use *irregular* and *dynamic* data structures (e.g. priority queues, hash tables and linked lists) for the storage of the search tree, as its size is not known at the outset. It grows exponentially with the size of the problems and remains often too large to be solved successfully as the program stops for lack of time or memory space. Parallelism is then used to speed up construction of search trees and to provide more memory space. The management of the irregular and dynamic data structures in parallel processing is therefore the key for high performance of computation.

Research during the last decade in the parallelization of B&B/A* algorithms showed that MIMD architecture is suitable. However, from experimental machines with exotic architecture to commercial distributed multiprocessors and networks of workstations, there still are two main high-level programming models : shared memory model (SMM) and distributed memory model (DMM). Vendors begin to propose parallel machines (KSR1, CRAY T3D) with programming environment allowing the two models.

In the SMM, data is stored in a global shared memory which can be accessed by all the processors. This model could be implemented on distributed memory machines via the *Virtual Shared Memory* mechanism (e.g. ALLCACHE system on KSR1). On the other hand, in the DMM, each processor can access only its local memory : data is exchanged by a message-passing mechanism (e.g. PVM, MPI). The problems involved by these programming models are completely different.

As data is shared in the SMM (section 2), the main issue is the contention access to the data. The higher the *concurrent access* to the data is, the higher is the performance. Data consistency is provided by a mutual exclusion mechanism, and load balancing is easy to achieve since data can be accessed by any processor.

On the other hand, *load balancing* is the key point to tackle in the DMM (section 3). As each processor can only access its own memory, if all the data have been processed in a local memory, the corresponding processor becomes idle while the other processors are still running.

In this paper, we propose new concurrent data structures and load balancing strategies for respectively the SMM and the DMM. In order to test their efficiency on the COP, we implemented them for B&B and A* algorithms to solve QAP, VCP and a 15 puzzle game.

2 Concurrent access to data structures

In the SMM, the data structure is stored in the shared memory. Each asynchronous processor must be able to access it. The simplest method to allow concurrent access to a data structure is that each asynchronous process locks the entire data structure whenever it makes an access or an operation. This exclusive use of the entire data structure to perform a basic operation serializes the access to the data structure and thus, limits the speed-up obtained with the parallel algorithm.

The next section presents a better technique which is denoted by **Partial Locking**.

2.1 Partial locking protocol

An operation is composed by successive elementary operations. At one moment, a processor only needs few nodes of the tree. Severity of contention can be reduced if each processor locks only the part of the data structure that it actually modifies. Hence, the time delay, during which the access to the structure is blocked, is decreased.

We reiterate the definition of *user view serialization* of the structure introduced by Lehman and Yao, 1981 [28], Calhoun and Ford, 1984, [5] in the context of data base management. In this approach, the processors that access the data structure with a well-ordering scheme, inspect a part of the structure that the previous processors would never change further, and leave all parts of the data structure (modified or not) in a consistent state.

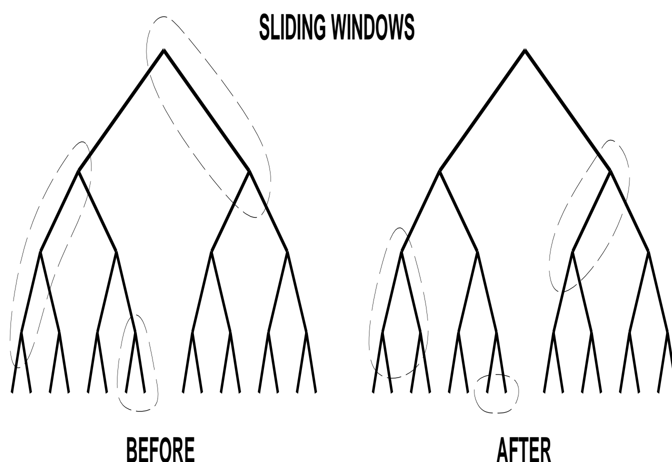

Figure 1: Example of applying the partial locking protocol to a tree data structure.

The figure 1 shows an example of this method applied to a tree data structure. A sliding window of a processor executing a basic operation on a tree data structure, is defined as the set of nodes of this data structure on which the processor has an exclusive access.

To forbid the creation of a cycle of processors waiting for resources (deadlocks), we have to force each processor to follow a specific locking scheme.

A simple scheme of this top-down method can be described as follows. Each operating window will be moved down the path, from the root to a leaf.
Notations: an ancestor of x is defined as a node which belongs to the (unique) path between x and the root.

Définition 1 *Let S a tree data structure in which all basic operations are executed downwards. The scheme is defined by the following rules:*

Lock *: a processor can request a lock on a node x only in the following cases*

- **1** *: if the processor has a lock on the father of x,*
- **2** *: if the processor has a lock on an ancestor of x and a lock on a descendant of x,*
- **3** *: a processor can lock the root r of S iff the processor has no more locks in S.*

Unlock *: no specific rule.*

The full description of the variant schemes and the proofs can be found in [6].

2.2 Partial locking boolean protocol

If the scheme designed above allows concurrent accesses, it implies an overhead due to the number of locks used : theses have an expensive execution time.

In respect to the locking scheme defined in 1, if a processor P_i has a lock on a node A of data structure S, P_i is the only processor which can request a lock on a node B, son of A (see figure 2).

Hence, expensive mutual exclusion primitives are not needed. Locks could be replaced by a boolean-like protocol. This is just a flag, which indicates if the node is in used or free. Since the root node has no father, several processors can try to access it. A lock will be necessary for this only node.

To illustrate this optimization, we test a Skew-heap using the partial locking paradigm [21] in the context of a simulation of B&B (figure 3) on the machine KSR1 with one processor. The execution time shows that the locks protocol overhead is really bigger than the boolean protocol overhead. Further, boolean protocol time is not very different than the serial time.

2.3 Related works

We can find in the literature two main kinds of techniques which allow "parallel" operations on data structures for shared memory machines. They are usually called *Parallel data structures* and *Concurrent data structures.*

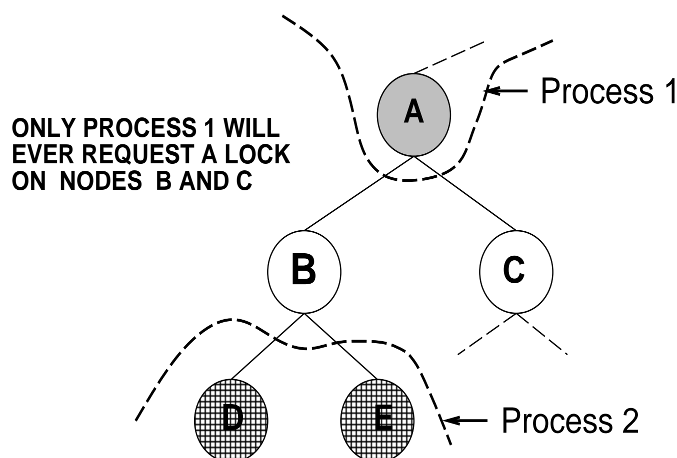

Figure 2: Only one processor can ever request a lock on a node.

Serial	Locks	Boolean
22 s	57 s	28 s

Figure 3: Times of Skew-heap to perform 200000 operations.

Parallel Data structures offer several synchronous operations performed by several processors. Processors cooperate to perform their operations [10, 11, 9]. Deo and Prasad in [11] have proposed an optimal parallel heap. P processors can simultaneously perform P operations. But these P operations are synchronous. Then, a processor who wants to access the data structure, has to wait the other processors. We do not want to use these kinds of parallel data structures, because in our applications the time between two accesses to the data structure is never the same on two processors. Some solutions use specialized processors to handle the data structure. We prefer to employ each processor used to execute the tree search algorithms

Concurrent data structure algorithms allow concurrent access to a data structure by asynchronous processors. Each processor performs a serial operation which is modified in order to keep the others processors unblocked. Processors make their operation themselves and do not need some other processors to access to the data structure. They do not cooperate to perform their operations. These methods are very useful, if the delay between two accesses is unknown. Therefore, mutual exclusion primitives are generally used to make atomic update operations with respect to the other processors. The partial locking protocols are such techniques. Initially the idea of partial locking can be found in articles dealing with data base management. Several solution had been proposed to allow concurrent access on Search Tree such as AVL-Tree, B-Tree and B+Tree [12, 13, 26]. Afterwards, this technique has been used for different priority queue such as D-heap [3, 38], Skew-heap [21], Fibonacci Queue and priority pool [19] . Sometimes, these concurrent

algorithms need specialized processors, to rebuild the data structure. Our contribution is an optimized version of a partial locking method. We have reduced the overhead due to the mutual exclusion primitives. For example, Jones' Skew-heap in [21] have $\log(n)$ mutual exclusion primitives calls for one operation. Our solution uses only one mutual exclusion primitive and $\log(n)$ boolean like protocol calls. Figure 3 shows the importance of this optimization.

In the next two sections, we show the efficiency of our partial locking boolean protocol applying it to data structures used in a parallel best-first B&B and in a parallel A* algorithm.

2.4 Priority queues for best-first B&B

Using the best-first strategy in B&B algorithms seems to be the most efficient, because the number of evaluated node is optimal. The data structure used in the best-first B&B is a priority queue (PQ). The basic operations are given in section 1.1.

In serial, PQ are usually represented by heap. There exist several algorithms which manage a heap : D-heap [22], leftist-heap [22], Skew-heap [47], Binomial queue [4], Pairing heap [15, 45]. The most popular one is the D-heap[2] as used in the heapsort ([49]). This is the oldest PQ implementation with $O(\log(n))$ performance. In a heap, the priority structure is represented as a binary tree that obeys the heap invariant, which holds that each item always has a higher priority than its children. In a D-heap, the tree is embedded in an array, using the rule that location 1 is the root of the tree, and that locations 2i and 2i+1 are the children of location i. In a B&B, each item contains an external pointer to store the subproblem information. Bistwas and Browne [3], Rao and Kumar [38] have proposed a concurrent version of the D-heap. Both have used a partial locking protocol. Our partial locking boolean protocol can not be used with the D-heap, because the operations of the algorithm need a direct access to the tree.

Serial experimental results [20, 6] show that the D-heap is not an efficient PQ implementation. Tarjan and Sleator proposed in [47] a better heap algorithm, the Skew-heap. Their algorithm is the self-adjusting version of the Leftist-heap. The basic operation which is used to implement $DeleteMin$ and $Insert$ operation is called the merge operation. Its *amortized complexity* is $O(\log(n))$. It seems that the Skew-heap is one of the most efficient serial algorithms for heap implementation ([20, 6]). Jones [21] have proposed a concurrent Skew-heap using a partial locking protocol with only mutual exclusion primitives. Applying our locking boolean protocol to the Skew-heap offers better performances (see figure 3).

The complexity of the heap $DeleteGreater$ operation is $O(n)$, because the tree must be entirely explored. An other problem is that heaps are not stable. In [31], Mans and Roucairol demonstrate that the PQ must be stable. The order of the nodes with the same priority must be fixed (LIFO or FIFO). This property avoids some anomalies of speed-up. Then, several PQ which are not a heap structure have been proposed. We can cite the

[2] Jones in [20] called it Implicit-heap.

funnels [30, 6] (table and tree) and the different Splay-trees [47, 20]

The funnel data structures have been developed by Mans and Roucairol in 1990 [30] for a best-first B&B implementation. However, they can be used in a lot of other applications. The B&B algorithms only need a small bounded interval of priority. The size of the interval is denoted by S. Initially S is equal to $ub - lb$ (ub is the cost of the best known solution and lb is the evaluation of the problem). During the execution of the algorithm, lb increases and ub decreases, then S decreases until it reaches 0. Then, there is a very simple way to achieve an efficient representation for the PQ. The idea is to use an array of FIFO of size S, where each FIFO j of the array is associated to a possible priority[3]. The serial experimental results show that the funnel table is the most efficient priority queue. Our partial locking boolean protocol can not be used to make concurrent the access to the funnel table, because it is not a tree data structure. All details about the funnel table can be found in [30, 6]. The funnel tree uses a complementary binary tree with S external nodes, where S is the smallest power of 2 which is greater than $ub - lb$. Each internal node of the tree has a counter which represents the number of subproblems being contained in the FIFOs below it. This tree is used to drive the $DeleteMin$ operation to the FIFO containing the best nodes. The complexity of operation is $O(\log(S))$. The funnel tree operations are made concurrent using our partial locking boolean protocol.

The Splay-trees initially are self-adjusting binary search tree. Jones in [20] shows that Splay-trees could be used as efficient priority queues. There exist several versions of serial Splay-tree data structures depending on the Splay operation : Splay, Semi-Splay, Simple-Semi-Splay. Each of them has a Top-Down and a Bottom-Up version. These different versions of Splay-trees support search operation on any priority. Then, we create new versions of Splay-trees (called Single) where each tree node is associated with a fixed priority value. We apply our partial locking boolean protocol to the Top-Down versions of the Semi-Splay, the Simple-Semi-Splay, Single-Semi-Splay and the Single-Simple-Semi-Splay.

The funnels and Splay-trees support efficient $DeleteGreater$ operations and have the stability property.

The figure 4 recapitulates the complexity and the properties of each priority queues.

PQ	Insert	DeleteMin	DeleteGreater	Perf	Stab
Funnel-Table	O(1)	O(1)	O(1)	1	Y
Splay	O(log n)	O(log n)	O(log n)	2	Y
Single-Splay	O(log S)	O(log S)	O(log S)	2	Y
Funnel-Tree	O(log S)	O(log S)	O(log S)	3	N
Skew-heap	O(log n)	O(log n)	O(n)	3	N
D-heap	O(log n)	O(log n)	O(n)	4	N

Figure 4: Summarizing of all priority queues.

[3] We make the assumption that priority value are integer.

2.4.1 Experimental results

We have tested these data structures on a best-first B&B algorithm solving the Quadratic Assignment Problem. The Quadratic Assignment Problem (QAP) is a combinatorial optimization problem introduced by Koopmans and Beckman, in 1957, [23]. It has numerous and various applications, such as location problems, VLSI design [36], architecture design [14],...

The objective of QAP is to assign n units to n sites in order to minimize the quadratic cost of this assignment, which depends both on the distances between the sites and on the flows between the units. We use the algorithm of Mautor and Roucairol [32]. Their algorithm uses the depth-first strategy, we modified it to use the best-first strategy.

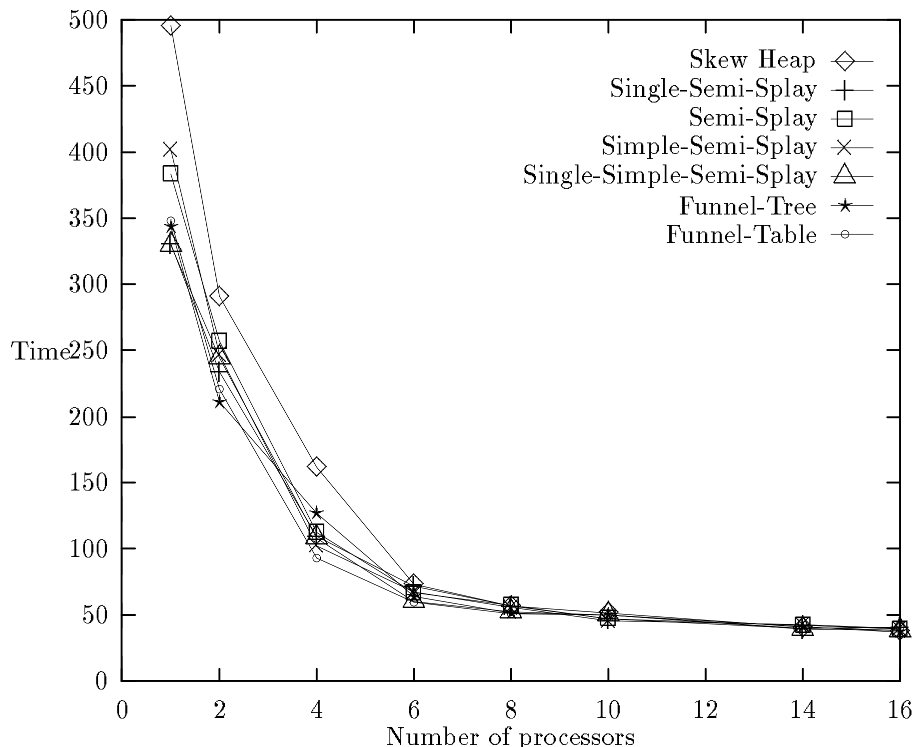

Figure 5: Time results on KSR1 machine solving the nugent 15 problem

Figure 5 shows the results obtained on a KSR1 machine with virtual shared memory. The programs solve the nugent15 problem which was obtained from QAP Library. We can see that the Splay-trees and the funnels PQ are most efficient than the Skew-heap until 6 processors. That confirms the theoretical complexity, and that the heap is not a good PQ representation for the best-first B&B. The single Splay-trees seem to be more efficient than the non single. The speed-up obtained is also very good until this limits. With more than 6 processors, the time does not decrease any more and the time of the different PQ are quite the same. This phenomenon is due to the problem size (96000 nodes are evaluated with the nugent15). If the size of the problem is small, the number of

nodes in the data structure is small, then the depth of the priority queue is small. Now, the number of simultaneous accesses depends on the depth of PQ. Thus, the speed-up is limited by the size of the problem. We began to test our program with bigger problem (nugent 16) to verify this conclusion. However, the time obtained with 6 processors is very good according to the machine and the problem.

2.5 Concurrent Treap for parallel A*

In this section, we present an application of the partial locking protocol (cf. section2.1) on a *double criteria* data structure called *treap.* This data structure is particularly suitable for the implementation of parallel A* algorithm because there is no cross-referencing between two data structures as we will see later. Therefore, in the data structure management we save memory space and avoid deadlock problems.

2.5.1 The A* algorithm

The space of potential solutions of a problem is generally defined in Artificial Intelligence 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 satisfy.

Generally, the state space is represented by a graph $G = (E, V)$, where E is a set of edges labelled by the operators and the costs of transformation, and V a set of nodes/states. To find an *optimal* solution of a problem is equivalent to exhibit a least cost path from the initial state to a goal state in the associated graph. The A* algorithm [35, 37] is generally used to find out such an optimal path.

On the theoretical level, the A* algorithm is a special case of the B&B paradigm. Like the B&B algorithm with the best-first strategy [34], an evaluation function $f = g + h$ is defined to give a priority between nodes. Thus, useless parts of the graph could be discarded and the *combinatorial explosion* reduced. The g function gives the cost of the path from the initial state to the current state and the h heuristic function estimates the cost of the path from the current state to a goal state.

When the A* algorithm is used to do a tree search on the state graph, it is similar to a best-first B&B algorithm. However, the A* algorithm is more difficult to deal with when it makes a graph search. In a graph search, a state could be reached by several paths with different costs. Therefore, a comparison mechanism must be added to find out whether a state is already explored or not and thus avoiding redundant work. Afterwards, we study a suitable data structure for the A* algorithm in graph search.

From an algorithmic point of view, the A* algorithm manages two lists, named OPEN and CLOSED. The OPEN list contains the states to be explored in the increasing order of

their priorities (f-values), while the CLOSED keeps the states already explored without priority order. The basic operations to access the OPEN list O are:

- $Search(O, state)$ finds the node x in O such that $x.state = state$;
- $Insert(O, x)$ adds the node x in O but $x.state$ must be unique in O;
  let y be a node already in O such that $y.state = x.state$,
  - if $y.priority < x.priority$, x is inserted, y is removed,
  - if $x.priority \leq y.priority$, x is not inserted;
- $DeleteMin(O)$ selects and removes the node x in O with the highest priority;
- $Delete(O, state)$ removes the node x from O such that $x.state = state$.

For the CLOSED list C, only three operations are necessary:

- $Search(C, state)$ finds the node x in C such that $x.state = state$;
- $Insert(C, x)$ adds the node x in C;
- $Delete(C, state)$ removes the node x from C such that $x.state = state$.

2.5.2 Towards a double criteria data structure

A simplest way to parallelize the A* algorithm, in the shared memory model, 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 is available for all processors [25] via the OPEN list. This scheme is also named centralized scheme. The main difficulty in this scheme is the management of the OPEN list in a concurrent environment.

Several parallelizations of the A* algorithm have been proposed for tree search, just like the best-first B&B algorithm (see [18] for a large survey). But at our best knowledge, there are only a few specific parallel implementations of the A* algorithm [25, 8]. The reason is that the A* algorithm seems to be difficult to parallelize [39]. The only work to do in parallel in A* is the management of OPEN and CLOSED global lists. Furthermore, no suitable concurrent data structures for the OPEN and CLOSED lists have been proposed for parallel formulations in the literature.

Priority queues [1, 44] are generally used until now and may be suitable for the *DeleteMin* operation according to the priority criterion (cf. section 2.4). But, they are completely inefficient for the *Search* operation with the state criterion. Other data structures such as hash tables, AVL-trees [1], Splay-trees [13, 47], etc, are efficient for the *Search* operation but not for *DeleteMin*. There were no data structures with *both operations* working on each 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 is not simple to achieve as we have seen in section 2.1.

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 deadlock problems and memory space overhead.

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

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

2.5.3 Treap data structure

Serial reap was introduced by Aragon and Seidel [2]. The authors use it to implement a new form of binary search trees : Randomized Search Trees. McCreight [33] uses it also to implement a multi-dimensional searching. He calls it *Priority Search Trees*[4].

Let O be a set of n nodes, a *key* and a *priority* are associated to each node. 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 O is a rooted binary tree with a node set O 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 subtree 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 node 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 (f-value of this state) are associated to each node. Thus, we can use a treap to implement the OPEN list of the A* algorithm.

Let T be the treap storing the node set O. The operations presented in the literature that could be applied on T are *Search*, *Insert* and *Delete*. We add one more operation *DeleteMin* and modify the *Insert* operation to conform to the basic operations of A* (cf. section 2.5.1).

Given the key of x, a node $x \in O$ can be easily accessed in T by using the usual search tree algorithm.

[4] 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 abandoned it in favor of the more commonly used *priority search tree* [33].

As several binary search trees [43, 47, 13, 26, 6], the update operations use a basic operation called *rotation* (figure 6).

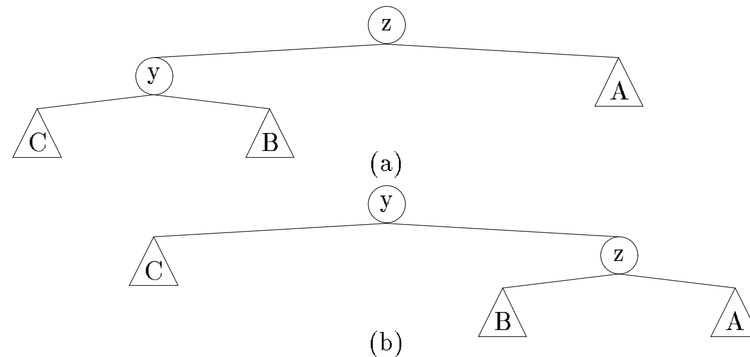

Figure 6: Rotation used to reorganized the treap.

In the literature, the $Insert$ operation works as follows. First, using the key of x, it attaches x to T in the appropriate leaf position. At this point, the keys of all the nodes of the modified tree are in In-order. To re-establish Heap-order, it simply rotates x as long as its parent has 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 a node x in T.

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

- a node y is found such that $y.priority < x.priority$ (cases (1) and (2) of Figure 7),
- a node z is found such that $z.key = x.key$ (case (3) of Figure 7).

If such a node z is found, the algorithm ends because it is guaranteed that $x.prority < z.priority$. Thus x must not be inserted in T. On the other hand, if such a node y is found, the node x is inserted at this position (between y and the father of y).

Let SBT_x be the subtree of T rooted in x. At this point, the priorities of all the nodes of the modified tree (SBT_x) are in Heap-order. To re-establish the In-order we use the splay operation [43, 47]. SBT_x is split in a Left subtree and a Right subtree. The left subtree (resp: right) contains all the nodes of SBT_x with the key smaller (resp: larger) than the key associated with x. Finally, the 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 a node y such that $y.key = x.key$, during the splay operation, the node y, is deleted (the y priority will be smaller then the 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

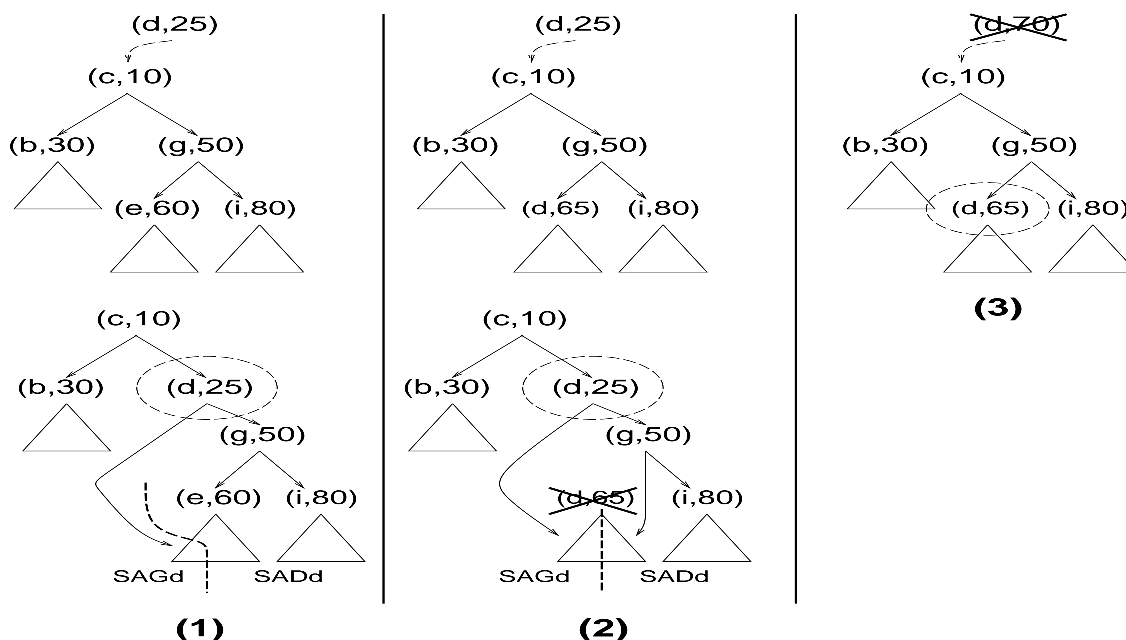

Figure 7: Insertion algorithm.

T such that $x.key = key$. Thus, first we search for such a node x and then 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 a key and a priority. Thus, the set occupies $O(n)$ words of storage. The time complexity of each operation is proportional to the depth of the treap T. If the key and the priority associated with a node are in the same order, the structure is a linear list. However, if the priorities are independent and identically distributed continuous random variables, the depth of the treap is $O(\log(n))$ (the treap is a balanced binary tree). Thus, the expected time to perform one of these operations, is still $O(\log(n))$ (n number of nodes in T) [33].

To get a balanced binary treap in an 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.

To allow concurrent access to the treap, we use partial locking boolean protocol described in section 2.2.

2.5.4 Experimental results

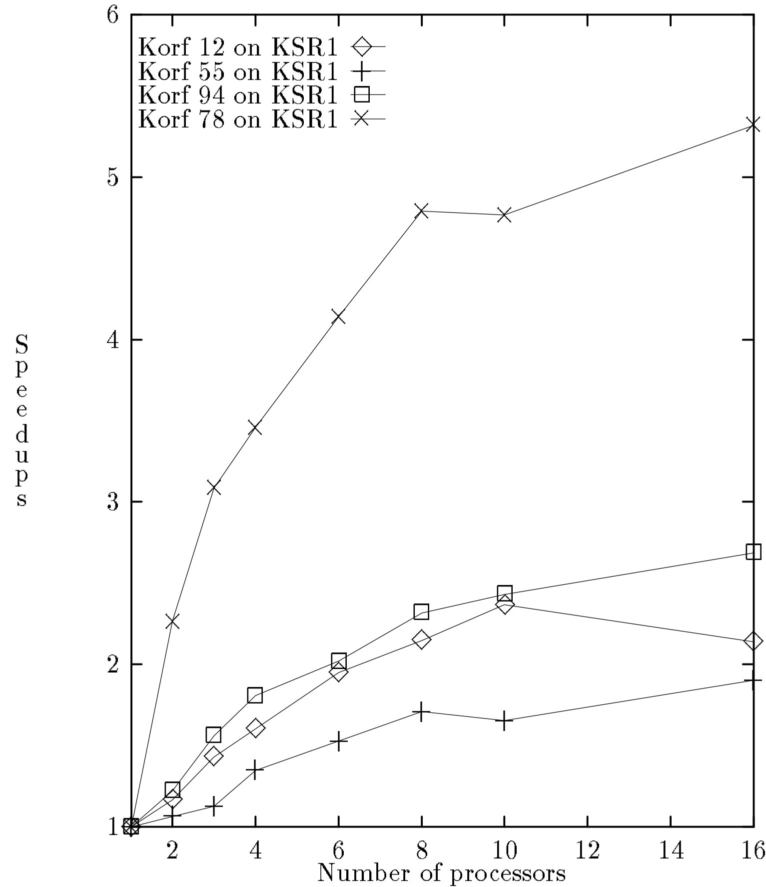

Figure 8: Results on KSR1 with some 15 puzzle games.

The implementation of this algorithm has been done on the shared memory architecture KSR1 [7].

Applied on some relatively small instances of the 15 puzzle problem proposed by Korf [24], we have compared these results (figure 2.5.4) with those obtained under the same experimental conditions, but with a combination of two data structures (a Skew-heap with a Splay-tree) for the OPEN list. The release with the combination of two data structures gives no speed-up. We have also compared with the results obtained by Rao, Kumar and Ramesh [39]. On the 15 puzzle game, their speedup reached a limit with less processors. Moreover, their implementation was done on a Sequent Balance 21000 which had a physical shared memory, the communication speed over computation speed ratio is greater on their machine than on the KSR1. Our data structures would then be

more efficient on the Sequent than on the KSR1. Therefore, we could already claim that the *concurrent treap* surpasses all the other data structures proposed for the centralized parallelization of A*.

Another on-going study is to apply this algorithm on other discrete optimization problems such as allocating processes on processors in order to dwindle communication cost.

3 Load balancing strategies

Load balancing is an important factor in parallel processing to dwindle processor idleness and optimize performance, especially in the DMM. With B&B algorithms, nodes are classified in function of their priorities into several queues, and solutions may be reached more efficiently if priorities given to nodes by queues are taken into consideration during the parallel execution of the algorithm.

Existing load balancing schemes use partial cost to balance nodes generated by best-first B&B distributed algorithms.

In this section, we show that using partial cost might lead to a big accumulation of high priority tasks on a few processors, while the other processors continue to work with low priority nodes. In such a situation, an other notion of priority is needed. It must take into account not only the partial cost of nodes but also their capacity to generate other nodes. Our aim is to develop an efficient load strategy that uses priority of tasks.

We have developed three load balancing strategies with this new notion of priority: an *one-by-one* and two versions of *partial distribution* strategies.

Our new notion of priority is applicable to various applications among which those using best-first B&B algorithms and those from the area of Operational Research. We have chosen to evaluate the strategy with a B&B solution of the Vertex Cover Problem (VCP).

The implementation of the B&B algorithm was carried out in a network of heterogeneous Unix workstations used as a single parallel computer through a Parallel Virtual Machine (PVM) software system. All experiments in this section have been done with 3 SUN workstations (1 SPARCstation ELC and 2 SPARCstations IPC).

3.1 The PVM environment

PVM [16] is a software system that permits a network of heterogeneous Unix computers (workstations in our case) to be used as a single large parallel computer (named the *virtual machine*). Thus, larger computational problems may be solved by using the aggregate power of many computers.

PVM supplies functions to automatically start up tasks on the virtual machine and allows tasks to communicate and synchronize with each other. PVM may start up more than one process on any processor in the PVM virtual machine. This affectation is done by calling the routine *pvm_spawn()*. A task is for us a Unix process.

Applications, such as B&B algorithms, can be parallelized by using message-passing constructs common to the most distributed-memory computers. By sending and receiving messages, multiple tasks of an application can cooperate to solve a problem in parallel.

Like that, PVM provides the flexibility to develop parallel applications using its communication facilities. The interested user can refer to [16, 46] for more details about PVM.

3.2 Distributed best-first B&B

We use our distributed version of best-fist B&B to solve the VCP.

3.2.1 B&B for VCP

For an undirected graph $G = (V, E)$ a subset $U \subseteq V$ of nodes is called a vertex cover if every edge is covered by a node from U, i.e. $\{u, v\} \in E$ implies $u \in U$ or $v \in U$. This problem aims to find the vertex cover with minimal cardinality.

All nodes are coded as tuples (SC, pc) where SC is a sub-cover (a set of nodes that cover some edges of the graph G) and pc is the number of nodes in SC; from now on, we call the **partial cost** of the treated node (lower bound on the sub-cover cardinality).

Our B&B algorithm generates two new nodes (SC_1, pc_1), (SC_2, pc_2) from a given node (SC, pc). The generation of SC_1 and SC_2 by the branch procedure is done by searching for a node v with maximal degree in the rest-graph (part of the original graph that has not yet been covered). $SC_1 = SC \cup \{v\}$, $SC_2 = SC \cup \{w \in E\}$ (for more details, see [29]).

3.2.2 Implementation

We have implemented the distributed B&B scheme of [29] with different strategies of load balancing as we will see later. Our aim is to develop an efficient load balancing strategy that uses priority of nodes. Therefore, the chosen algorithm is sufficient for our purpose (for more information about B&B algorithms and their parallelizations, refer to [40, 27, 17, 48, 50]).

The chosen B&B algorithm uses a distributed queue organization to temporarily stock the generated nodes. We chose to execute only one process per processor in the PVM machine. Each processor has a local queue in its local memory and executes the sequential B&B algorithm using this queue.

In the best-first search strategy, nodes are classified in many queues in decreasing order of their priority. The evaluation of a node is given by its partial cost.

With the distributed B&B scheme, the communication plays an essential role in determining the performance especially when we use multi-user machines (like workstations). To improve performance, we must try to reduce the number of exchanged messages during the execution, as we will show in subsequent sections.

3.3 Motivations and justifications of our approach

We have developed three load balancing strategies, an one-by-one and two versions of partial distribution, to understand load distribution between the different slaves. We started by testing all of these strategies with the generated nodes of each processor classified into a local queue according to their associated **partial cost**.

In the one-by-one load balancing strategy, each slave needing work receives only one node from the beginning of the queue of another loaded processor (figure 9 shows an execution for a graph of 75 vertices with average degree of 50%).

Figure 9: The load distribution by "on_by_one" load balancing strategy.

In the partial distribution strategies, the number of tasks received by a processor requiring work is variable. The number of nodes sent from the maximum loaded slave to the one requiring work is equal to the minimum of the average load and the local load minus the average load.

	Slave_0	Slave_1	Slave_2	Slave_3	Slave_4
Local load	10	0	8	17	19
Average load = 54/5 # 10 tasks per slave.					
Deviation	0	-10	-2	5	9
		min. deviation			max. deviation

Send min(Average load, (local load - average load))

Figure 10: An example of application of partial distribution load balancing strategy, where slave_1 asks the other slaves for work.

The first version of partial distribution load balancing strategy performs load balancing by sending nodes from the beginning of the local queue whereas the second one sends nodes from the end of the queue.

3.4 Results with classical priority notion

We have noticed that we get a very bad distribution of load in the case of the one-by-one strategy; at the beginning all slaves have the same quantity of work but after that one of them (the second slave) becomes strongly loaded although this slave is not the one where the root node in the B&B tree has been assigned.

We can explain this situation by the fact that some nodes potentially generate more work than the others in the search space.

We have noticed also that with version 2 of the partial distribution load balancing strategy, the loads of different slaves are globally more equally distributed comparing to the version 1, and that with version 2 load balancing operations occur less frequently than with version 1. This proves that the **partial cost** used as a criterion to classify nodes within the queues is not the only parameter to apply to get a good load balancing strategy.

The above results indicate that it is important to take into consideration the potential work that nodes can generate.

Any efficient load balancing strategy should ensure, as much as possible, that the processing of nodes occurs in the global order of their partial cost and should take into consideration their associated potential work. Consequently, we need to define a priority notion between different nodes which takes into consideration these two parameters.

3.5 A new notion of priority : potentiality

A node generated during the execution of B&B algorithm has two associated values : its cost and an indicator representing the potential work it can generate successively to reach an optimal solution (see figure 11). These two values depend on the treated problem. In the VCP for example, the first one is equal to the partial cost, and the other value will be taken equal to the number of edges in the rest-graph.

Each generated node is characterized by its position into the indicated region in figure 7. The execution of B&B algorithm starts from **Start** point (where we have the root of B&B tree) and terminates at **End** point (the searched optimal solution).

The potentiality of a node sp is defined as follows :

Définition 2

$$Potentiality(sp) = (dist(sp, End), potential_work, ancestors_nb).$$

Where :

sp *is the considered node,*

$dist(sp, End)$, *the distance between the position of sp and the* **End** *point,*

$potential_work$, *the value of the associated potential work of sp,*

$ancestors_nb$, *the number of the ancestors of sp.*

In this definition of priority, we introduced a third parameter, the number of ancestors, to select between nodes that have the same distance from the End point and the same quantity of potential work. We chose to privilege those having the less number of ancestors.

In practice, the value of the optimal solution is not known before the end of the execution, but usually we start execution with an upper bound that maximizes this value.

In the VCP, for example, we can use the number of nodes of a given graph as an End point and we compute distances relatively to it.

Generated nodes are classified into the different queues during the execution of B&B distributed algorithm as a function of their priorities. A node sp_1 has more priority than another node sp_2, if it verifies one of the following rules:

$$dist(sp_1, End) < dist(sp_2, End) \tag{1}$$

$$\begin{gathered}(dist(sp_1, End) == dist(sp_2, End))\&\& \\ (sp_1.potential_work > sp_2.potential_work)\end{gathered} \tag{2}$$

$$\begin{gathered}(dist(sp_1, End) == dist(sp_2, End))\&\& \\ (sp_1.potential_work == sp_2.potential_work)\&\& \\ (sp_1.ancestors_nb < sp_2.ancestors_nb)\end{gathered} \tag{3}$$

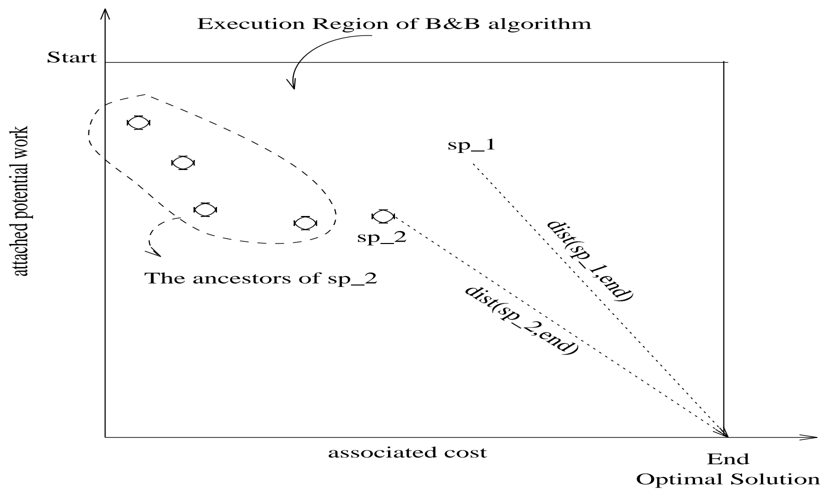

Figure 11: Definition of priority of two nodes sp_1 and sp_2.

3.6 Load balancing strategies with potentiality

This notion has lead to improvements such as a better load distribution among the different processors in the PVM machine, and a reduction of the number of exchanged messages during the execution.

3.6.1 Influence on load distribution

For the one-by-one load balancing strategy, we remark that at the beginning of the execution, the assigned nodes have more priority than the others so that the number of load balancing operations is lower than its number without using our priority notion. At the end, we have a lot of load balancing operations when the priority of nodes begins to decrease (figure 12 with the same graph as figure 9).

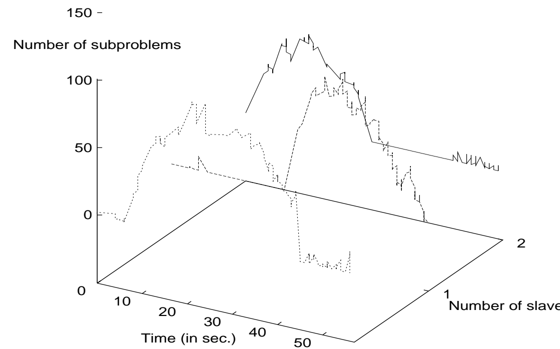

Figure 12: The load distribution by "one_by_one" strategy using the notion of potentiality.

The load is equally distributed with partial distribution load balancing strategies (version 1 and 2).

We can see also that the load of the most loaded slave does not exceed the 110 nodes while it was around 450 nodes (with version 1) and 600 nodes (with version 2), without using our potentiality notion.

3.6.2 Influence on communication costs

To show the importance of the communication factor, we have chosen to study the exchanged messages between the master and all of its slaves.

For the one-by-one strategy, we have remarked that there is a clear difference in the number of sent and received messages with and without using the potentiality notion. This number is lesser in the first case (figure 13).

For the two versions of partial distribution load balancing strategy, we have remarked that the number of sent and received messages takes an asymptotic value. In version 1, the number of exchanged messages remains approximately the same, while in version 2 the utilization of our priority notion lightly increases the number of exchanged messages. In fact, with this version, when an unloaded slave takes nodes from the end of the queue of a loaded slave, this unloaded slave executes the work and rapidly becomes unloaded. This light augmentation of the number of load balancing operations shows that nodes are really classified in queues as a function of their priority of execution.

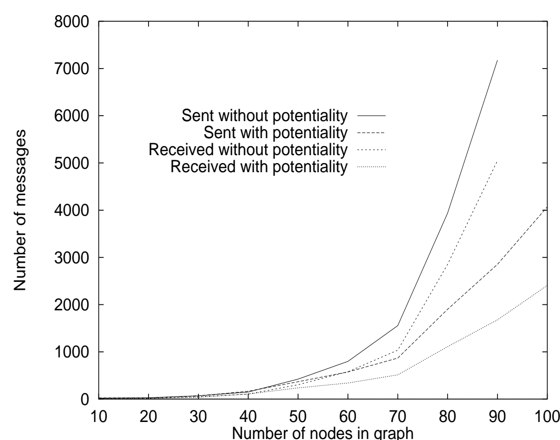

Figure 13: The sent and received messages using "one_by_one" strategy with and without the notion of potentiality.

4 Concluding remarks & perspectives

For the SMM, we have proposed a set of concurrent data structures suitable for Tree/Graph search algorithms such as best-first B&B and A* algorithms. Parallel best-first B&B needs efficient concurrent PQ. Theoritically, the Funnel and the Splay-tree priority queues are more efficient than the classical Heap or Skew-Heap. The experimental results, obtained on the KSR1 machine with a parallel B&B solving the Quadratic Assignement Problem, seem to confirm this fact. The problem of the partial locking method is that the number of simultaneous access is limited by the depth of the tree. But as much as we know, if we want to keep the strict best first strategy and to allow asynchronous access, it seems that the partial locking method is the only solution.

Our contribution is an optimized version of the partial locking protocol in the case of tree structures called the *partial locking boolean protocol*. We have reduced the number of mutual exclusion primitive calls. Thus, the overhead is reduced. We plan to test our concurrent priority queues to solve bigger instances of the QAP and other problems such as TSP, Graph coloring.

In this model, we have also presented a double criteria data structure : the *concurrent treap*, 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. We have implemented concurrent access for these operations with the technique of partial locking.

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. This data structure could be also used in other applications in which the management of two criteria is needed.

Results obtained on the KSR1 and applied to some 15 puzzle games show that the concurrent treap is efficient for a group of ten processors. The speedups presented are

better than those in [39] with the same parallel scheme.

For the DMM, we analyzed the behavior of three load balancing strategies and proved the necessity to have a new priority order between the different executed nodes in a B&B application.

We introduced the notion of potentiality based upon an estimation of the capacity of a node to generate work, and on an evaluation of the distance between the node and the optimal solution. We believe that this kind of potentiality can be applied to any B&B application where nodes are characterized by the previous parameters.

Furthermore, we studied the impact of this notion on the behavior of load balancing strategies for distributed best-first B&B applications. These strategies show that after taking into consideration the potentiality of nodes, their selection to participate to an operation of load balancing is not the same at the beginning and at the end of the execution.

All these load balancing strategies are developed in a centralized manner (there is one master processor that controls the other ones in the PVM machine). Future works will consist of investigating the effect of the use of our potentiality notion when several masters instead of only one (like the strategies in [42, 41]) are used.

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. Bitwas and J. Browne. Simultaneous update of priority structures. *IEEE international conference on parallele computing*, pages 124–131, 1987.

[4] M. Brown. Implementation and analysis of binomial queue algorithmms. *SIAM Comput.*, 7:298–319, 1978.

[5] 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.

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

[7] V.-D. Cung and B. L. Cun. A suitable data structure for parallel a*. RR 2165, Institut National de Recherche en Informatique et Automatique (INRIA), Jan. 1994.

[8] 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.

[9] S. Das and W.-B. Horng. Managing a parallel heap efficiently. In *Proc. PARLE'91-Parallel Architectures and Languages Europe*, pages 270–288, 1991.

[10] N. Deo. Data structures for parallel computation on shared-memory machine. In *SuperComputing*, pages 341–345, 1989.

[11] N. Deo and S. Prasad. Parallel heap: An optimal parallel priority queue. *Journal of Supercomputing*, 6(1):87–98, Mar. 1992.

[12] C. Ellis. Concurrent search and insertion in 2-3 trees. *Acta Informatica*, 14:63–86, 1980.

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

[14] A. Elshafei. Hospital layout as a quadratic assignment problem. *Operational Research Quarterly*, 28:167–179, 1977.

[15] M. Fredman, R. Sedgewick, D. Sleator, and R. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1:111–129, 1986.

[16] A. Geist, A. Beguelin, W. Jiang, R. Manchek, and V. Sunderam. *PVM 3 User's Guide and Reference Manual.* Oak Ridge, Tennessee 37831, May 1993.

[17] B. Gendron and T. G. Crainic. Parallel branch-and-bound algorithms: Survey and synthesis. Technical Report 913, Centre de recherche sur les transports, Montréal (Canada), May 1993.

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

[19] Q. Huang. An evaluation of concurrent priority queue algorithms. In *Third IEEE Symposium on Parallel and Distributed Processing*, pages 518–525, Dec. 1991.

[20] D. Jones. An empirical comparaison of priority queue and event set implementation. *Comm. ACM*, 29(320):191–194, Apr. 1986.

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

[22] D. Knuth. *The Art of Programming: Sorting and Searching*, volume 3. Addison-Wesley, 1973.

[23] T. Koopmans and M. Beckman. Assignment problems and the location of economic activities. *Econometrica*, 25:53–76, 1957.

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

[25] 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.

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

[27] E. L. Lawler and D. E. Wood. Branch and bound methods: A survey. *Operations Research*, (14):670–719, 1966.

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

[29] R. Lüling and B. Monien. Two strategies for solving the vertex cover problem on a transputer network. In *proc. of the 3rd Int. Workshop on Distributed Algorithms*, pages 160–171. number 392 in Lecture Notes of Computer Sciences, 1989.

[30] B. Mans and C. Roucairol. Concurrency in priority queues for branch and bound algorithms. Technical Report 1311, INRIA, 1990.

[31] B. Mans and C. Roucairol. Performances des algorithmes branch-and-bound parallèles à stratégie meilleur d'abord. RR 1716, INRIA-Rocquencourt, Domaine de Voluceau, BP 105, 78153 Le Chesnay Cedex, 1992.

[32] T. Mautor and C. Roucairol. A new exact algorithm for the solution of quadratic assignment problems. *Discrete Applied Mathematics*, to appear, 1993. MASI-RR-92-09 - Université de Paris 6, 4 place Jussieu, 75252 Paris Cédex 05.

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

[34] 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.

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

[36] C. Nugent, T. Vollmann, and J. Ruml. An experimental comparison of techniques for the assignment of facilities to locations. *Operations Research*, 16:150–173, 1968.

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

[38] 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.

[39] 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.

[40] C. Roucairol. Parallel branch & bound algorithms - an overview. Technical Report 862, INRIA-Rocquencourt, Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France), 1989.

[41] V. A. Saletore and M. A. Mohammed. Hierarchical load balancing schemes for branch-and-bound computations on distributed memory machines. In *Hawaii International Conference on System Software*, Jan. 1993.

[42] A. B. Sinha and L. V. Kale. A load balancing strategy for prioritized execution of tasks. In *Workshop on Dynamic Object Placement and Load Balancing in Parallel and Distributed Systems (in co-operation with ECOOP '92)*, June 1992.

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

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

[45] J. Stasko and J. Vitter. Pairing heap: Experiments and analysis. Technical Report 600, I.N.R.I.A., Feb. 1987.

[46] V. S. Sunderam, G. A. Geist, J. Dongarra, and R. Manchek. The pvm concurrent computing system: Evolution, experiences, and trends. *Parallel Computing*, 20(4):531–546, March 1994.

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

[48] B. W. Wah and C. F. Yu. Stochastic modeling of branch-and-bound algorithms with best-first search. *IEEE Transactions on software engineering*, SE-11(9):922–934, September 1985.

[49] J. Williams. Algorithm 232: Heapsort. *CACM*, 7:347–348, 1964.

[50] M. K. Yang and C. R. Das. Evaluation of a parallel branch-and-bound algorithm on a class of multiprocessors. *IEEE Transactions on parallel and distributed Systems*, 5(1):74–86, January 1994.

Laboratoire PRiSM, Université de Versailles - Saint Quentin en Yvelines,
45 avenue des État-Unis, 78035 Versailles Cedex, FRANCE.